# JITNIC: Just-in-Time Compilation for In-Network Processing

*Samwise Thomas Parkinson*

Master of Science

Ann S. Bowers College of Computer and Information Science

Cornell University

2022

## Abstract

With the end to Moore's Law looming ever closer, it is becoming ever more crucial to design and effectively leverage specialized hardware components. This has taken shape in the domain of networking in the form of programmable switches and new domain-specific languages used to configure such devices. However, a large portion of the packet-processing workload is still handled by general-purpose hardware on servers despite rapid advances in the efficiency and versatility of these network devices.

This paper presents JITNIC, a new formalism and corresponding software prototype, which applies methods from dynamic compilers to leverage the speed and flexibility of programmable network hardware. This novel application of just-in-time compilation defines a distributed implementation of the semantics of packet-processing programs, allowing a general-purpose CPU to delegate the execution of program traces onto the network card. After developing the theoretical foundations for just-in-time compilation for NICs, the implementation is experimentally validated in a prototype setting, producing results that indicate potential for significant speed-up in a practical system.

## Acknowledgements

I have the most sincere gratitude for the guidance and mentorship of my advisor and committee chair Professor Nate Foster, whose insight, knowledge, and wisdom was indispensable throughout the process of writing this thesis and developing the supporting body of work. Many of my skills that led to the success of this project I owe at least in part to his instruction. Special thanks also go out to Professor Dexter Kozen, who guided my supporting coursework in mathematics, and Professor Adrian Sampson, who first introduced me to the concept of a just-in-time compiler. Finally, I wish to thank my colleagues, Ryan Doenges, Yunhe Liu, Mark Moeller, and Rudy Peterson, all of whom assisted in various stages of the project by providing revisions for the document, contributing feedback to the supporting math and code, suggesting related reading, and much more.

## Biographical Sketch

Samwise Thomas Parkinson is currently in his 2nd year of study at the Ann S. Bowers College of Computer and Information Science at Cornell University, focusing on programming languages with applications in the networking domain. He will graduate with a Masters of Science in August of 2022. Previously, he graduated with a Bachelor of Arts from the Cornell College of Arts and Sciences in May of 2020, double majoring in philosophy and computer science, minoring in Spanish, and focusing primarily on mathematical logic and programming languages.

Before his first degree, Samwise spent his first two summers on a service trip and study abroad in Madrid before settling on academic research as a career interest. His following summers were spent in research and software engineering on a project studying P4, a new programming language in the networking domain. He also spend a considerable portion of his time making novel contributions to course materials for a number of high level classes in the computer science department as a teaching assistant, for which he received awards from the department on several occasions.

# Table of Contents

# Chapter 1

# Introduction

In 1965, Gordon Moore predicted that innovation in electrical engineering would allow the transistor-density of computer hardware to approximately double every year [Moore, 1965]. This famous extrapolation, now known as Moore's Law, has held true over several decades due to the continuous cutting-edge advances in technology and the dedicated efforts of innovators around the globe. In recent years, it has become a popular trend to incorrectly predict the end of Moore's Law, with experts heralding the doom of fast-paced advancement in transistor technology. Of course, this is not entirely unwarranted. Transistor technology has certainly faced serious challenges as our circuits shrink to the microscopic level.

However, there are serious concerns for the rate of hardware speed-up. Moore himself knew this, and made several qualifications to his initial prediction. He has since clarified that his 1965 statement was intended to refer only to the next 10 years [Moore, 2015]. Moreover, he revised his prediction in 1975, declaring that transistor density should be expected to double every 18 months, not every year [Moore, 1975]. It is now quite common to see Moore's Law stretched or misquoted as the prediction that transistor density should double every 2 (or sometimes even 2.5) years [Kelleher, 2022], lending additional leeway to the pace of technological advancement. Again, this is not done without reason, as it is becoming more and more difficult to dismiss the idea that the rate of increase for hardware efficiency is starting to slow down [Hennessy and Patterson, 2019]. There is even a well-known theoretical limit on how small transistors can be. Active research has been able to demonstrate the potential feasibility of single-atom or even singe-electron transistors, but any smaller is infeasible due to the unpredictability of quantum mechanics. For reference, a single-electron transistor can be as small as 1.5 nanometers [Cheng et al., 2011], while the transistors

in the newest iPhones today are only 5 nanometers wide [Apple, 2022].

In light of these challenges, chip manufacturers have begun turning to areas of innovation besides more efficient CPU components to accelerate processing. Notably, advancements in specialized hardware have enabled computers to perform restricted sets of tasks at rates that are orders of magnitude faster than those of a traditional CPU. Advances in quantum computing have made an entire collection of NP-hard problems feasible to solve in polynomial time, FPGAs provide the flexible circuitry that enables the prototyping, testing, and verification of new circuits at an extremely fast pace, and new network hardware components are able to perform advanced cryptography and hashing algorithms at line rate in the network.

As innovation in specialized hardware continues, we must also develop new systems in order for users to leverage their maximum potential. Historically, this happened as computing technology evolved from hard-wired configuration of mainframes computers to the modern PC. As the gate-count of processors grew, more user-friendly ways were needed to configure them efficiently and correctly, leading to the development of high-level programming languages and compilers. Likewise, more sophisticated processors placed higher demand on firmware and operating systems for security and efficiency, leading to advances in memory management and distributed computing. In general, we find that as hardware components grow in complexity and efficiency, so do the systems that maintain, manage, and utilize them.

In recent years, this same pattern is emerging in various domains with new, specialized hardware components. Quantum algorithms for solving NP-hard problems in polynomial time have been well-understood for several years in anticipation of the advent of quantum computing. Hardware description languages such as Verilog have a vast assortment of supporting and related tools for testing, verifying and even automatically synthesizing circuits to leverage flexible prototyping environments. Network programming languages such as NetKAT and P4 enable the testing, verification, and deployment of flexible configurations onto specialized network hardware.

*Just-in-time Compilation.* One such approach that has grown quite pervasive over the last 10-20 years is the just-in-time (JIT) compiler. Traditionally, high-level programming languages are implemented by defining a translation from the high-level, human-readable format of C and Java to a minimal instruction set implemented by the hardware, such as MIPS and x86. This approach is convenient for large programs that change infrequently and run often, and it opens the door to a wide array of static analyses and optimizations that can provide strong security and efficiency guarantees.

The main drawback of this approach, however, is that code must be re-compiled each time a change is made. This can be expensive for large systems, and it can slow down the process of development. Alternatively, languages might be implemented by an interpreter, as in the case of Python and Javascript. An interpreter is a pre-compiled software system that directly implements the semantics of the high-level language. Interpreters are typically inefficient in their executing developer code, but they save the overhead of re-compiling each time a change is made. This makes them most convenient for prototyping and testing smaller systems, but less convenient for the full-scale development and deployment of software.

A JIT is a system that combines both of these approaches to provide a language implementation that reacts to its workload at runtime, providing dynamic optimization. Starting with a standard interpreter, a JIT tracks data about the program over multiple executions. Using this data, it identifies fragments of the program and compiles them in the usual way. During future executions, the JIT dispatches to the compiled fragment when possible and falls back to the main interpreter in the general case. A JIT typically maintains multiple compiled program fragments. Using metadata about the program at runtime, it will decide when to introduce freshly compiled components and when to evict stale ones. This approach allows a language implementation to provide the low-overhead response of an interpreter combined with the optimizations and efficiency of a compiler. Moreover, a JIT provides the additional benefit of an adaptive speedup that adjusts in to the workload in real time. The main drawbacks are the memory and runtime overhead needed for profiling the program over multiple executions.

*Software Defined Networking (SDN).* At the same time as the emergence of JIT methods, the fabric of network configuration has been rapidly and radically transforming. With the rise of data centers as the backbone of the internet, demand has increased for network devices that are flexible and conveniently configurable. Network devices used to be hardwired for a small collection of protocols. Now, they have become configurable almost to the same degree as general purpose CPUs, even though they are fundamentally different in architecture and support highly restricted instruction sets. The newer, more flexible hardware makes room for new high-level programming languages, such as NetKAT and P4, to target network devices such as routers, network interface cards (NICs), and network controllers.

The main strength of network devices is their ability to select between a large collection of actions at incredible speed. This is achieved by means of the routing table (or match-action table), a hardware component that reads incoming data from incoming

packet headers and selects the appropriate action to perform. A typical example would consist of parsing the packet's destination from the packet header and sending the packet out the appropriate port, often adjusting metadata such as timestamps or logs on the way out. However, as software-defined networking makes networks more flexible, and as network hardware becomes more sophisticated, both the information found in packet headers and the actions to which routing tables dispatch become more interesting and complex.

*Contributions.* In this thesis, I explore a new method that leverages the efficiency and flexibility of specialized network hardware to speed up general processing in the CPU. I draw on methods from tracing JIT compilers and apply them in the domain of network programming. The result is a new formalism and corresponding software system that serves as a template for developing a JIT compiler. Given a CPU program that processes data on a per-packet basis, the JIT will identify "hot" paths of execution, compile them to a network programming language, and deploy them on the Network Inteface Card (NIC) with corresponding path conditions. I leverage the traditional routing table on the NIC to dispatch to the pre-compiled traces at runtime and fall back to the general-purpose machine when needed. In the best case, this will allow a high percentage of packets to be processed entirely in the network, eliminating the server entirely.

The key hurdle with this novel approach to packet processing is the issue of memory consistency. JIT compilers are normally designed and implemented to execute on a single machine. In this setting, incoming packets will be processed on two separate devices that implement the program in a distributed manner, and there are many pitfalls to navigate. It is important to ensure that when the output of the program will be affected by loads from memory, loads are executed at runtime from up-to-date memory values on both the NIC and the CPU. As an additional challenge, it is a question of open research how to implement distributed memory for distributed packet processing on the CPU and NIC, and a variety of viable approaches exist. To resolve these issues, I develop a formalism that abstracts over the implementation of distributed memory and identifies the weakest preconditions under which the behavior of each program trace remains the same. As a result, JITNIC provides a template for deploying a tracing JIT that correctly implements the semantics of packet processing programs regardless of the distributed memory model, and it leverages the specialized hardware of a network card to provide significant speed-up that adapts to the workload at runtime.

The rest of the paper organized as follows. Chapter 2 gives more detailed background

on both JIT methods and SDN tools with working examples. Chapter 3 introduces a formalism for mapping an execution path of a standard program to its path condition and an equivalent partial program that can be deployed on a network device. Chapter 4 introduces a software system that implements the formalism in OCaml. Chapter 5 provides evaluation of both 3 and 4. Chapter 6 deals with limitations and future and related work, and Chapter 7 concludes.

# Chapter 2

# Background

JITNIC is a formalism and software tool that builds on methods from JITs and applies them to leverage the speed and flexibility of network hardware. Therefore, I provide background in two parts. First, I describe the state-of-the art in just-in-time compilers, focusing on tracing JITs and using an example for exposition. Next, I give background in software-defined networking, focusing on the fundamental features of networking most relevant to the design of JITNIC and introducing the source and target languages of the dynamic compiler.

## 2.1 Just-in-Time Compilation

As I have previously discussed, a JIT compiler (or dynamic compiler) is a system that implements the semantics of a high-level programming language by executing an interpreter and dynamically identifying fragments of the program to partially or totally compile. If execution invokes a fragment that is already compiled, the JIT halts execution of the interpreter, executes the compiled fragment (the "fast path"), and returns to normal execution of the interpreter. Otherwise, the normal interpreter is used by default (the "slow path"). In practice, a program "fragment" is either a method or a trace of execution. It is common to categorize JITs as either *method* JITs, in the case of the former, or *tracing* JITs in the case of the latter. The key distinction between the two approaches is that a method JIT flags methods for compilation that are frequently invoked and/or are costly to execute, whereas a tracing JIT does the same for traces of execution through the control flow, possible spanning many methods. JITNIC most resembles a tracing JIT, so I will focus mostly on the tracing style of JIT. However, most features of dynamic compilers are common to both styles.

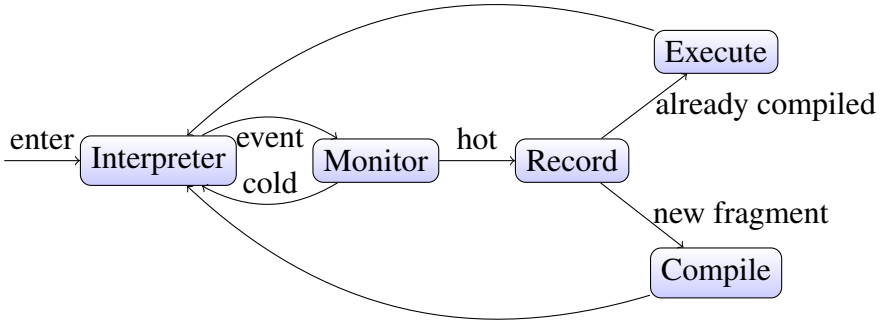


Figure 2.1: The components of a JIT and the transitions taken during execution. Execution of a JIT starts in a standard interpreter. An event, either a method invocation or a branch in control flow, causes a change to the monitoring information. The monitor then decides if the current program fragment is "hot" or "cold". For a cold fragment, the JIT returns to normal execution in the interpreter. Conversely, the JIT consults the record on a hot fragment. If a compiled version of the fragment already exists, it is invoked. Otherwise, the fragment is compiled and added to the record. Finally, execution continues in the interpreter.

*How to JIT.* Figure 2.1 illustrates the states and transitions that a JIT may take during execution. It the initial configuration, a JIT behaves as a normal interpreter maintaining a monitor. The monitor is a data structure that tracks various program fragments based on certain criteria, usually frequency of invocation. Once the frequency of a given program fragment reaches a certain threshold, the monitor identifies it as "hot" and passes it to the compiler. The JIT keeps a record of compiled program fragments, and will dispatch to them in future executions as they are needed.

*Example.* Consider the program in Figure 2.2, which gives a simple recursive implementation of a Python program that finds the $n$-th number in the Fibonacci sequence. In order to design a tracing JIT for this program and walk through its execution on two different inputs, a few choices about the configuration must be made. For simplicity, I will only allow the JIT to record one program trace at a time, and traces will be restricted to a single method call. The criterion for moving a trace into the record will be if it has been executed more than any other observed trace. If two traces have been observed the same number of times, the trace that has been seen most recently will be used. I will walk through the execution of the program on input $n = 4$ and explain what the JIT does at each step in execution. In this example, there are only two possible traces and corresponding path conditions: the `if n <= 2` clause and the `else` clause. I will refer to these as Trace #1 and Trace #2, respectively.

```
1 def fibn(n):
2     if n <= 2:
3         return 1
4     else:
5         return fibn(n-1) + fibn(n-2)
```

Figure 2.2: A recursive implementation of the Fibonacci sequence in Python

On the input $n = 4$, the JIT starts with an empty monitor and record, and begins the interpreter execution of `fibn(4)`. Since the input is greater than 2, the interpreter enters Trace #2, a new trace. The JIT therefore adds an entry to the monitor for this new trace and initializes the counter for this trace to 1. At this point, Trace #2 has been executed the most, so the JIT compiles the code for Trace #2 and adds the compiled trace to the record. Next, it proceeds by executing the call `fibn(3)`. The interpreter now enters Trace #2 a second time, incrementing the counter to 2 and making use of the fast, pre-compiled copy in the record. Then, it makes the supporting calls `fibn(2)` and `fibn(1)`, both of which enter Trace #1. This adds Trace #1 to the monitor, and the counter reaches 2. Now, Trace #1 has been seen more recently and as many times as Trace #2. This meets the conditions to evict the compiled copy of Trace #2 from the record and replace it with a compiled copy of Trace #1. Finally, when the last call `fibn(4-2)` from the first execution of the method gets executed, the JIT sees that it already has a compiled version of Trace #1 in the record, so it may use the fast path for the final step.

This example illustrates the ability of a JIT to optimize in reaction to its workload. By the second execution of Trace #2, there is already a fast copy of the program fragment. Afterwards, the program has no need for the fast copy of Trace #2, and it is able to detect this and compile a fast copy of Trace #1 in time for the final execution of the method to take the fast path. Of course, a production-grade just-in-time compiler operates on a much larger scale. It will have the resources to store a much larger number of pre-compiled program fragments in the record, and it will make use of much more sophisticated data structures and heuristics for moving fragments in and out. Moreover, the program fragments it deals with will be much larger in size and complexity.

*Why JIT?* This complex methodology, large memory overhead, and costly monitoring raises the natural question: is it really worth it to just-in-time compile programs? Put another way, is it really the case that pre-compiled program fragments offer enough speed-up and get invoked frequently enough to justify the overhead? The answer is

somewhat more nuanced than a simple "yes." Certainly the compiled fragments execute orders of magnitude faster than in a normal interpreter. The main difficulty lies in the task of configuring the rules for monitoring and selecting fragments for compilation is a challenging one, and there is a great volume of engineering research on the subject. Once a proper configuration is identified, however, JIT implementations provide the flexibility and low overhead of an interpreter with significant speedup. As a result, dynamic compilers have proved to be a powerful tool for programming language implementation, so much so that a great many have been deployed for large-scale use in a variety of settings (see Chapter 6).

## 2.2 Network Programming

Because JITNIC is a system that applies JIT methods in the domain of networks and network programming, some basic background on the fundamental features of networking is needed. In this section, I briefly review some basic components of network design such as packets, routing tables, and network design patterns. I also introduce the source and target languages of JITNIC, eBPF and P4.

*Packets and Routing.* A network is a collection of devices, or *switches*,[1] whose purpose is to provide data sharing between various endpoints, or *hosts*. When Host A has data to send to end point B, it divides the data into smaller pieces, called *packets*. Host A prepends some metadata to each packet, called the *header*. The header contains data about the source, destination, and lifetime of the packet. The packets traverse the network travelling from switch to switch, and finally arrive at Host B, whose role it is to reassemble the packets into the original data. Various protocols exist for generating and interpreting the data in packet headers for the purposes of forwarding the packets to the correct destination, detecting and recovering from data loss, and reconstructing streams of packets into large data at the endpoints.

Switches within the network must be able to receive a packet, read the information in the header, and perform a small action on the packet. For example, a router typically reads the source destination from the header and chooses the next intermediate destination for the packet. Similarly, a firewall might read the source address from the packet header and decide whether to allow the data through. The demand placed on

[1] The word switch in the context of networking has been used with a wide variety of definitions in mind. For simplicity of exposition, I use it here to refer to any device connected to the network that operates on the data-plane of the network.

such devices is that they should be able to receive, process, and transmit individual packets "at line rate," that is to say, at the same speed as bits on a wire. Of course, any reading, modifying, or monitoring of data will incur some cost. However, advances in specialized hardware continue to drive the rate of packet processing closer and closer to line rate. As a result, switches are typically able to perform orders of magnitude faster than general-purpose hardware, but are only able to perform a restricted set of tasks.

To achieve this goal, almost all network devices make use of a hardware component called a *table* (also called a *routing table*). A routing table dispatches packets header to "actions" based on values found in specific fields of the packet header. A table *entry* consists of a key and an action. The key is a set of values accepted from a given field in the packet header, such as source/destination. Actions are simple units of computation, such as marking to drop the packet or setting the destination for the packet's next hop.

Tables are configured by a *controller* – a central device[2] that maintains a picture of the network and decides how to direct flows of data traversing the network. In a standard model, each network devices keeps connectivity with the controller and provides information about itself. For tables, this means that a given switch defines the structure of its tables to the controller. This includes what values from the packet header it can match on, what actions it can perform, how many entries it has space for at a time, and what the default action is. In response, the controller issues commands related to the entries in the table, populating the table with entries and removing outdated ones. In short, the controller gives each device rules for processing packets, and the devices apply these rules efficiently to each individual packet.

*eBPF Language.* The Berkeley Packet Filter (BPF) language is a minimal hardware language resembling x86. Originally, its purpose was to provide a way to send small programs across the network that execute on the destination device. Because of the limitations of this usage, and because of the obvious security concerns, BPF is a highly restricted instruction set whose implementation is a virtual machine running in the kernel. Most notably, efficient static analyses on the VM ensure that the BPF programs are memory safe and terminate. The successor language, eBFP (extended BPF) is an extension of BPF mainly used for processing data on a per-packet basis. It is usually compiled from C programs and it is implemented on a similar VM.

*P4 Language.* Controllers typically resemble standard processors. They make use of sophisticated data structures to maintain an updated picture of the network and measure

[2]This is not strictly true in many networks. Controllers, while conceptually central to the dynamic configuration of the network, are often implemented in a distributed fashion across multiple devices.

traffic between various pairs of end points. They also implement computationally complex algorithms, such as Dijkstra's shortest path algorithm, to make decisions about how to install and remove table entries on connected switches. Consequently, they are often configured with standard programming languages such as Python and C. In contrast, the hardware of network devices is highly specialized, and is structured differently from a normal processor. Therefore, a specialized language is needed to configure a network device.

P4 is a high level language designed for configuring network devices. Syntactically, it resembles loop-free C, but it includes domain-specific features such as table declarations and primitives for invoking specialized hardware components. A program in P4 defines a packet-processing pipeline with various components, and the program will be applied to each packet. Figure 2.3 gives an example P4 program that parses the Ethernet and IPv4 headers from the packet header, applies a table to the destination address of the packet, and emits the packet with updated values in the header. Lines 14-18 demonstrate how the language allows the programmer to define the parameters of a table while still allowing the controller to populate the entries of the table in the standard manner. Also, lines 20 and 22 demonstrate how the programmer may invoke primitive operations supported by the hardware. The exact semantics of the language depend heavily on the implementation and on the target device. In particular, (1) the set of primitive hardware operations supported, (2) the structure of the pipeline, and (3) the semantics of transitioning from one stage of the pipeline to the next are all variable and heavily dependent on the target device. While this may seem like a drawback, the ambiguity allows for great versatility; P4 may be implemented for and deployed on a wide variety of network devices.

```
1  parser Parse(packet_in p, out headers h, inout metadata m) {
2      state start {
3          p.extract(h.ethernet);
4          p.extract(h.ipv4);
5          transition accept;
6      }
7  }
8  control Ingress(inout headers h, inout metadata m) {
9      action forward(bit<9> port) {
10         m.egress_spec = port;
11         h.ipv4.ttl = h.ipv4.ttl - 1; //decrement time to live
12     }
13     table ipv4 {
14         key = { h.ipv4.dstAddr: exact; }
15         actions = { forward; drop; }
16         default_action = drop();
17     }
18     apply {
19         verify_checksum(h.ipv4.hdrChecksum);
20         ipv4.apply();
21         update_checksum(h.ipv4.hdrChecksum);
22     }
23 }
24 control Deparse(packet_out p, in headers h) {
25     apply {
26         p.emit(h.ethernet);
27         p.emit(h.ipv4);
28     }
29 }
30 Switch(Parse(), Ingress(), Deparse()) main;
```

Figure 2.3: An example P4 program. Lines 1-7 define a packet parser that extracts the first few bytes of the packet into P4 data structures representing Ethernet and IP headers. Lines 8-23 define the main packet processing component, with a forwarding action defined on lines 9-12 and a match-action table defined on lines 13-17. The apply block on lines 18-22 is the block of code that gets executed during packet processing, and it invokes the table on line 20 as well as specialized hardware components on lines 19 and 21 for hashing the packet to detect corrupted data. Lines 24-28 define a deparser that emits the (possibly modified) packet data structures ahead of the rest of the packet. Finally, line 30 packages all three components together into the main packet-processing pipeline.

# Chapter 3

# JITNIC Formalism

JITNIC is a system that applies methods from dynamic compilers in the domain of network programming, and it compiles eBPF programs to P4 programs to leverage the speedup that network hardware provides. This section provides a solid theoretical foundation for the implementation by working with minimal subsets of the source and target languages. First, 3.1 gives the semantics of the minimal subsets and describes the distributed JIT model in greater detail. Then, 3.2 gives symbolic evaluation semantics that produce an optimized trace and a corresponding path condition, shows how to map the trace onto a pipeline with three routing tables, and describes how to install corresponding table entries that implement the path condition. Whereas 3.2 gives a solution that assumes the source program only interacts with memory at most once, 3.3 discusses alternative approaches for generalizing the single-memory approach to traces with arbitrarily many memory operations.

## 3.1 Semantics & Distributed Model

In this section, I design minimal subsets of the source and target languages. The goal of this exercise is to define as simple a semantics as possible while still capturing what is semantically interesting or challenging about the languages for the purpose of building a tracing JIT. This allows me to reason about the formal properties of the tracing JIT unencumbered by repetitive syntax and tedious case analysis in inductive proofs. At the same time, conclusions made about the formalism provide a reasonable degree of certainty that the same properties hold in the context of the full languages.

Figure 3.1 defines syntax for the minimal subset languages. A command $c$ is a program in the source language, a minimal subset of eBPF. It includes standard control

$$n \in \mathbb{Z},\ x \in \mathit{Var} \cup \{\texttt{in1}, \texttt{in2}, ...\} \cup \{\texttt{out1}, \texttt{out2}, ...\}$$

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
e & ::= & n & a & ::= & \texttt{skip} & c & ::= & \texttt{skip} \\
& | & x & & | & x := e & & | & x := e \\
& | & \sim e & & | & [e_1] := e_2 & & | & [e_1] := e_2 \\
& | & e_1 \odot e_2 & & | & x := [e] & & | & x := [e] \\
& & & & | & a_1;\ a_2 & & | & c_1;\ c_2 \\
& & & & & & & | & \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2
\end{array}
$$

Figure 3.1: Syntax for expressions, actions, and commands.

flow such as the branching and sequencing, but omits loops because of the restrictions placed on eBPF implementations. It also includes base commands for working with variables and for reading and writing to memory. An action $a$ is an action in the sense of routing tables. It has all the components of the command language $c$ with the exception of conditional branches, because the goal of the JIT is to extract a straight-line trace of $c$ during execution. Actions and commands share the same expression language, which allows for integers $n$, variables $x$, and abstract unary and binary operators $\sim$ and $\odot$. The set $\mathit{Var}$ is assumed to contain a pre-defined collection of variable names along with the special variables $\texttt{in}_1$, $\texttt{in}_2$, ..., $\texttt{out}_1$, $\texttt{out}_2$, ... that are used to measure the input-output behaviour of actions and commands. Unary and binary operators are left undefined, but must not have side effects. They will typically allow standard arithmetic and boolean operators found in most hardware languages, such as boolean negation and integer addition. However, the no-side-effects constraint disallows division as a valid binary operator. Also, the absence of side-effects makes the semantic question of short-circuiting boolean operators irrelevant.

The distributed model of JITNIC implements the interpreter and monitor components of a JIT on the main device. The record component, however is implemented on the NIC A table $t$ will define its keys, actions, and default action. For simplicity, variables will be used as keys. A table entry consists of a logical predicate over the values of the keys and an action $a$. The pipeline will consist of 3 tables, and is designed around a single memory operation whose distributed implementation is left abstract. The first and last tables perform memory-free actions, while the middle table performs actions that consist of a single memory operation (see Figure 3.2). In this model, the first table configures the environment in anticipation of a load or store in the second

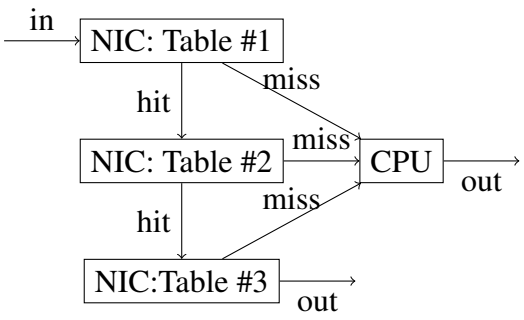


Figure 3.2: The workflow of the distributed JIT at runtime. On a new packet, the network device attempts to execute a pre-compiled trace. If no such trace exists, or if a semantically significant change in the memory is detected since the trace was compiled, the JIT falls back to the main packet-processing program. After the "out" from the CPU, the usual steps of a JIT (monitor, compile, record) are performed.

table. Then, the third table processes the results, producing the outputs. This allows the pipeline to check at three different points during execution that the inputs and memory satisfy the weakest preconditions of the original program trace. Should any of the checks fail, the entire execution gets discarded along with any side effects related to memory, and the JIT falls back to the slow path.

Full small step semantics for commands and actions are given in Appendix A, along with the proofs for some standard properties such as termination and determinism. The signature of the small-step relation for commands is

$$\langle \sigma, \varepsilon, c \rangle \to \langle \sigma', \varepsilon', c' \rangle,$$

and the relation for actions is identical. The multi-step relation $\to^*$ extends the small-step relation $\to$ in the standard way, so that $\to^*$ denotes the reflexive and transitive closure of $\to$. The store $\sigma$, representing memory, is a partial map from memory locations to values $\sigma : \mathbb{Z} \rightharpoonup \mathbb{Z}$. The variable environment $\varepsilon$ is likewise a partial map from variables to values $\varepsilon : \mathit{Var} \rightharpoonup \mathbb{Z}$. As a consequence of the way these data structures are defined, memory locations and values inhabit the same datatype in this formalism, as is the case in many low-level languages. The semantics for commands update and read from the variable environment and the store in the expected way. Expressions $e$ are used for the guard of if-else branches in place of including an additional boolean expression sub-language. Here, integers are used in the same way as in most low-level languages, with a value of 0 for `false` and any other integer value for `true`. The tracing JIT design is less concerned with the semantics of expressions, but they are still needed. I

therefore save notation by using big-step semantics with the signature $\langle \varepsilon, e \rangle \Downarrow n$, whose full definition is also given in Appendix A. Expression evaluation need not concern itself with memory, since expressions are not permitted to have side-effects.

## 3.2 Symbolic Evaluation

With these semantics in place, the next step is to extend the small step semantics with additional components that enable the generation of a straight-line trace that may be installed on the packet-processing pipeline. This will draw on techniques from symbolic evaluation. The goal is to define a way of partially evaluating a branching command $c$ exactly as much as is needed and no more. Specifically, the judgement must eliminate all branches from the command and identify the path condition without introducing additional assumptions that are immaterial to the path taken, simplifying as much of the program as possible along the way.

To achieve this, I define another small step relation over commands. It takes as input a command $c$, the program inputs in the form of a variable environment that binds only the input variables $\texttt{in}_n$, and the initial memory configuration $\sigma$. As output, it will produce an action $a$ to install on the pipeline, along with the path condition $\varphi$. For notational convenience, $\varphi$ is given in the form of value from the shared expression language $e$. As in the semantics, $\varphi$ is considered "true" if and only if it does not evaluate to 0 under the initial variable bindings for the input variables using the big step semantics for expressions. In this context, expressions $\varphi$ can be interpreted as predicates over the input variables. More generally, they can be interpreted as predicates over variable environments so long as the variable environment is defined only on the input variables (possibly including the special input variable $\texttt{in}_0$ for values read from memory, see below). I say that $\varepsilon$ *satisfies* $\varphi$, denoted $\varphi(\varepsilon)$, if $\langle \varepsilon, \varphi \rangle \Downarrow n$ and $n \neq 0$.

I introduce the notion of a *symbolic value* $\nu$,

$$\nu ::= \alpha \mid n \mid \sim \nu \mid \nu_1 \odot \nu_2,$$

which is a value whose precise integer value may or may not be fully known at runtime. A symbolic value is either an integer $n$, a symbolic variable $\alpha_n$, or a unary or binary operation applied to another symbolic value. A *symbolic variable* is a means of abstracting away part of the evaluation by leaving an un-evaluated hole in the program. For my purposes, it will suffice to use symbolic variables for the input values and values read from memory only, since these are the only values that can affect the path. A

consequence of this choice is that symbolic values $\nu$ can be used as normal expressions as long as it is clear to what variable each $\alpha_n$ corresponds. The *symbolic environment* $\mathcal{E} : Var \rightharpoonup \nu$ replaces the variable environment $\varepsilon$ by mapping variables to symbolic values rather than concrete values. Analogously, replacing the big-step judgement for concrete expression evaluation is the relation

$$\langle \mathcal{E}, e \rangle \downarrow \nu,$$

for symbolic evaluation of the shared expression language. The full set of inference rules for this judgement can be found in Appendix B.

The signature for the initial attempt at symbolic evaluation semantics will be

$$\langle \sigma, \varepsilon, c, a, \varphi, \mathcal{E} \rangle \Rightarrow \langle \sigma', \varepsilon', c', a', \varphi', \mathcal{E}' \rangle,$$

using the double right arrow since this is a small-step style relation with additional components. With the memory store $\sigma$ and the command $c$ as before, the intuition is that the action $a$ has new base actions appended as each base command gets executed, and the path condition $\varphi$ is a conjunction of branch conditions that gets extended once each conditional guard gets resolved. The concrete variable environment $\varepsilon$ remains, but will no longer be updated in the usual way. Instead, it serves as a record of the initial mapping of input variables for the rare cases in which concrete evaluation is needed. Once again, the multi-step relation $\Rightarrow^*$ is used to denote the reflexive and transitive closure of $\Rightarrow$.

Closer inspection of the formalism reveals, however, that this signature does not quite suffice. With two actions to be performed before and after a memory operation in the packet-processing pipeline, the path condition must be checked in two separate phases. With the above signature, it is unclear how to divide the output path condition $\varphi$ into two separate conditions. Therefore, I modify the above signature to the following:

$$\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle \Rightarrow \langle \sigma', \varepsilon', c', a', \varphi_1', \varphi_2', b', \mathcal{E}' \rangle.$$

In this modified judgement, the original path condition is split into two so that $\varphi = \varphi_1 \ \& \ \varphi_2$. The indicator $b$ switches from `false` to `true` as soon as the first memory operation is executed. At this point, new conjuncts extending the path condition stop extending $\varphi_1$ and start extending $\varphi_2$. The result is a simple method for splitting the path condition into table entries for the first and third tables.

With all of the machinery in place, I may now give a full definition for the small-step symbolic evaluation judgement. For example, the rule for symbolically evaluating an assignment command is as follows:

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \mathcal{E}' = \mathcal{E}[x \mapsto \nu]}{\langle \sigma, \varepsilon, x := e, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, \texttt{skip}, a;\ x := \nu, \varphi_1, \varphi_2, b, \mathcal{E}'\rangle.}$$

This simple step performs symbolic evaluation on the expression $e$ to update the symbolic environment, and then appends the assignment to the action for later use on the pipeline. The symbolic expression evaluation of $e$ doubles as an optimization step, since it is the simplified $\nu$ that gets used in the action.

Another interesting base command is the load from memory:

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \langle \varepsilon, \nu\rangle \Downarrow n \quad \varepsilon' = \varepsilon[\texttt{in}_0 \mapsto \sigma(n)] \quad \mathcal{E}' = \mathcal{E}[x \mapsto \alpha_0] \quad \texttt{in}_0 \textit{ is fresh}}{\langle \sigma, \varepsilon, x := [e], a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon', \texttt{skip}, a; x := [\nu], \varphi_1, \varphi_2, \texttt{true}, \mathcal{E}'\rangle}$$

This is the only case in which the variable environment $\varepsilon$ gets updated. This is because the value loaded from memory has the capacity affect the path taken through the program, and therefore behaves as an addition input to the rest of the program. It is for that reason that I add the concrete value read from memory to the fresh input variable $\texttt{in}_0$ and a corresponding symbolic variable $\alpha_0$ to the symbolic environment. Also, the judgement flips the indicator $b$, since a load constitutes a memory operation. This will cause the small-step relation to get stuck after observing multiple memory operations, but this is not a problem for now under the assumption that the program only interacts with memory at most once.

Branching commands are more complex and require multiple different inference rules for a proper characterization of the intended semantics. Here is once such rule:

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \langle \varepsilon, \nu\rangle \Downarrow n \quad n \neq 0 \quad \varphi_1' = \nu \ \texttt{\&\&}\ \varphi_1}{\langle \sigma, \varepsilon, \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f, a, \varphi_1, \varphi_2, \texttt{false}, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, c_t, a, \varphi_1', \varphi_2, \texttt{false}, \mathcal{E}\rangle}$$

This judgement produces a new path condition that is the old path condition in conjunction with the simplified conditional guard expression. Here, some concrete evaluation is needed to ensure that the correct branch of the conditional is selected. Also, in this case it is $\varphi_1$ that gets updated since the indicator $b$ is `false`. Had $b$ been `true`, $\varphi_2$ would have received the update.

The addition of these overhead values in the judgement along with the mixture of partial and concrete evaluation introduce concerns for the correctness of the symbolic

evaluation semantics. One would hope that symbolic evaluation of a command $c$ producing an output action $a$ followed by the concrete evaluation of $a$ has the same semantics as simple concrete evaluation of the original command $c$. This is captured in the following theorem:

**Theorem 3.1 (Semantics Preservation).** *Given a memory configuration $\sigma$, an $\varepsilon$ that is defined only on the input variables, and a command $c$. Let $\mathcal{E}$ be the initial symbolic environment defined only on the input variables, so that $\mathcal{E}(\texttt{in}_n) = \alpha_n$ for all $\texttt{in}_n$ defined in $\varepsilon$. Then if $\langle \sigma, \varepsilon, c\rangle \to^* \langle \sigma_1, \varepsilon_1, \texttt{skip}\rangle$, if $\langle \sigma, \varepsilon, c, \texttt{skip}, 1, 1, \texttt{false}, \mathcal{E}\rangle \Rightarrow^* \langle \sigma_2, \varepsilon_2, \texttt{skip}, a, \varphi_1, \varphi_2, b, \mathcal{E}'\rangle$, and if $\langle \sigma, \varepsilon, a\rangle \to^* \langle \sigma_3, \varepsilon_3, \texttt{skip}\rangle$, then $\sigma_1 = \sigma_3$ and $\varepsilon_1$ agrees with $\varepsilon_3$ on the output variables.*

The full proof of this theorem is given in Appendix B as a relatively straightforward (though non-trivial) induction on the structure of $c$. This theorem is not concerned with $\varepsilon_2$ because the symbolic evaluation relation does not attempt to produce concrete output. It also fails to make any claims about the path conditions, which is the purpose of the next theorem:

**Theorem 3.2 (Precondition Soundness).** *Given a memory store $\sigma$, an $\varepsilon$ defined only on the input variables, and a command $c$. Let $\mathcal{E}$ be defined as in Theorem 3.1, and let $\langle \sigma, \varepsilon, c, \texttt{skip}, 1, 1, \texttt{false}, \mathcal{E}\rangle \Rightarrow^* \langle \sigma_1, \varepsilon', \texttt{skip}, a, \varphi_1, \varphi_2, b, \mathcal{E}_1\rangle$. Then for any $\varepsilon_0$ that satisfies $\varphi = \varphi_1 \,\&\, \varphi_2$, if $\langle \sigma, \varepsilon_0, c, \texttt{skip}, 1, 1, \texttt{true}, \mathcal{E}\rangle \Rightarrow^* \langle \sigma_2, \varepsilon'_0, \texttt{skip}, a', \varphi'_1, \varphi'_2, b, \mathcal{E}_2\rangle$, then we have $a = a'$, $\varphi_1 = \varphi'_1$, and $\varphi_2 = \varphi'_2$.*

Theorem 3.2 captures the intuition that if a certain input generates a trace and a path condition, then any input satisfying that path condition would have done the same. This is important in the context of JITNIC because it often happens that two packets are very different, yet are similar enough to be processed by the same trace of the main program. In this case, the tracing JIT provides a much stronger optimization guarantee if the NIC table is general enough to implement a check that captures this notion of "sufficient similarity."

With the correctness and generality guarantees in place, all that remains is a small post-processing step. The so-called "action" produced by the symbolic evaluation phase is not in the proper format to be installed on the NIC's packet-processing pipeline as defined above. This is simple enough to remedy; Figure 3.3 gives a basic recursive method for breaking apart an action $a$ into three pieces and mounting them onto the pipeline described in the previous section.

MOUNT(a)

$$
\begin{aligned}
\textsc{Mount}(\texttt{skip}) &\triangleq \langle \texttt{skip}, \texttt{skip}, \texttt{skip} \rangle \\
\textsc{Mount}(x := e) &\triangleq \langle x := e, \texttt{skip}, \texttt{skip} \rangle \\
\textsc{Mount}(x := [e]) &\triangleq \langle \texttt{skip}, x := [e], \texttt{skip} \rangle \\
\textsc{Mount}([e_1] := e_2) &\triangleq \langle \texttt{skip}, [e_1] := e_2, \texttt{skip} \rangle \\
\textsc{Mount}(a_1;\ a_2) &\triangleq \begin{cases} \langle a_{1a}, a_{1b}, a_{1c}; a_2 \rangle & \text{if } a_{1b} \text{ is a load or store} \\ \langle a_1; a_{2a}, a_{2b}, a_{2c} \rangle & \text{if } a_{2b} \text{ is a load or store} \\ \langle a_1; a_2, \texttt{skip}, \texttt{skip} \rangle & \text{otherwise} \end{cases} \\
& \quad \text{where } \textsc{Mount}(a_1) = \langle a_{1a}, a_{1b}, a_{1c} \rangle, \\
& \quad \text{and } \textsc{Mount}(a_2) = \langle a_{2a}, a_{2b}, a_{2c} \rangle
\end{aligned}
$$

Figure 3.3: A simple method for partitioning a trace into actions that may be installed on the three-stage pipeline described previously. The middle action consists of a single memory operation, while the first and last actions do any other necessary computation.

Defining table entries from the path conditions is a much simpler task, since it is trivialized by the definition for a table in this formalism. Of course, standard hardware implementations of match-action tables cannot implement generalized logical predicates, and more work must be done to lift this formalism to a functional system, which I handle in Chapter 4. Notwithstanding, entry keys defined as logical predicates over variables allows, $\varphi_1$ to be installed on the first and second tables as the table entry key, and $\varphi_2$ may be installed as the entry key for the third table. Putting everything together, if $\langle \sigma, \varepsilon, c, \texttt{skip}, 1, 1, \texttt{false}, \mathcal{E} \rangle \Rightarrow^* \langle \sigma', \varepsilon', \texttt{skip}, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle$ and $\textsc{Mount}(a) = \langle a_1, a_2, a_3 \rangle$, then the above three theorems allow for $\langle \varphi_1, a_1 \rangle$ to be installed on the first table, $\langle \varphi_1, a_2 \rangle$ to be installed on the second table, and $\langle \varphi_2, a_3 \rangle$ to be installed on the third table. This will implement the correct semantics of the program trace in the restricted format of the packet processing pipeline defined in Section 3.1, as assured by Theorem 3.4:

**Theorem 3.3.** *If* $\textsc{Mount}(a) = \langle a_1, a_2, a_3 \rangle$*, then* $a_1$ *and* $a_3$ *contain no memory operation while* $a_2$ *consists of a single load or store, or is* `skip`*. Moreover,* $a$ *is semantically equivalent to* $a_1;\ a_2;\ a_3$*, that is to say,* $\langle \sigma, \varepsilon, a \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$ *iff.* $\langle \sigma, \varepsilon, a_1;\ a_2;\ a_3 \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$.

## 3.3 Handling Multiple Memory Operations

With a full formalism in place that handles traces with a single memory operation, I now turn my attention to programs with arbitrarily many memory operations. Of course, a JIT that only has the capacity to speed up programs that interact with memory at most once is not a system worthy of deployment in a practical setting. Therefore, I discard the assumption from the previous section that the command $c$ only interact with memory at most once. This done, I will briefly discuss two approaches to generalizing the symbolic evaluation approach from the previous section.

*Relying on hardware.* First, it is worth noting that the structure of the packet-processing pipeline used for the symbolic evaluation is overly simplistic and somewhat artificial. With rapid advances in network hardware, new packet-processing devices with increasingly diverse packet-processing pipeline are constantly being made available. In fact, this is a key motivator for the design of P4 as a flexible language with the expressiveness to configure pipelines of nearly arbitrary structure. Moreover, the flexibly configurable hardware of FPGAs allows for the rapid development of experimental hardware prototypes. In the case of JITNIC, this means there is no reason to assume strict constraints of the target pipeline. The three-stage pipeline with the $(compute; memory; compute)$ structure could be replaced with a pipeline that allows for arbitrarily many memory operations. For example, one might implement a four-stage pipeline for $(compute; memory; compute; memory; compute)$, or even a more specific $(compute; load; compute; store; compute)$, depending on the expected needs of the device and the network at large. In the case, the JITNIC formalism serves as a design template. All that is needed is to (1) further divide the path condition $\varphi = \varphi_1 \,\&\, \varphi_2$ into $\varphi = \varphi_1 \,\&\, ... \,\&\varphi_n$, (2) replace the boolean indicator $b$ with a counter, (3) adjust the small step relation accordingly, and (4) modify the MOUNT method. Refer to Appendix C for a concrete example of this process.

*Relying on packet re-submission.* An alternative approach to generalizing the single-memory formalism involves leveraging the `resubmit` operation that is standard to most networking devices. The goal is to install a trace with an unknown number of memory operations by breaking it into multiple trace fragments that fit the original three-stage, single-memory operation pipeline. By resubmitting a given packet multiple times, and by maintaining the proper metadata during a re-submit, this allows the minimal pipeline to implement arbitrary program traces in an iterative fashion. Normally, this approach would be a cause for concern, since a large volume of resubmitted packets

during in-network processing can fill buffers quickly, causing latency and even high rates of packet loss for multiple end-points at once. However, JITNIC is designed to be deployed on the NIC of a host. Therefore, only the host machine pays the penalty of multiple resubmits, and the speed-up of exporting packet-processing to the NIC more than justifies the trade-off in the expected case (see Chapter5).

Formally, this approach replaces the path conditions $\varphi_1$ and $\varphi_2$ and corresponding indicator with a stack of path conditions,

$$s ::= \langle\rangle \mid \varphi :: s$$

On each branch, the judgement extends the head of the stack by extending the previous head with the branch condition with a conjunction in the usual way. On a memory operation, it pushes a new condition to the head of the stack rather than flipping the indicator as before (full details in Appendix C). The end result is an ordered list of path conditions, each corresponding to a unique memory operation in the trace. The MOUNT method is extended to iteratively break apart the resulting action $a$ until no more memory operations are found. Finally, a special, hidden input variable is added to the program to count how many times a given packet has been resubmitted to the pipeline. The table entries refer to this input to dispatch to the correct fragment of a given trace so that the program gets executed in the proper order.

# Chapter 4

# Implementation

The JITNIC software system is implemented in OCaml in approximately 3400 LOC according to the `cloc` tool. It is a prototype environment that implements the small step relations for the minimal formalism described in Chapter 3. OCaml representations are defined for expressions $e$, commands $c$, symbolic values $\nu$, etc. as presented in Chapter 3. The implementation includes small-step interpreters for the source and target languages, an implementation of the symbolic evaluation JIT, and a test harness that checks the correctness of the JIT on various example programs. In the current state, the code only provides support for single-memory traces, analogous with the assumption from Chapter 3.2. The test harness mirrors the structure of the proofs in the previous chapter. It provides a collection of example programs and executes the following assertions on each example:

- it executes the small-step interpreter for the source language and compares the results to expected values,
- it executes the symbolic evaluation semantics and compares the results to the values produced by the small-step interpreter,
- it generates a pipeline from a single execution of the symbolic evaluation, and checks that the pipeline applied to the same input both (1) executes the pipeline without falling back to the slow path and (2) produces the same output as the symbolic evaluation,
- it generates a pipeline from a single execution of the symbolic evaluation, and checks that the pipeline applied to a different input with the same expected path of execution satisfies (1) and (2) from the previous bullet, and

- it generates a pipeline from a single execution of the symbolic evaluation, and performs a static check ensuring that the generated pipeline satisfies the structural constraints outlined in the previous chapter.

With an example set of 6 programs running on a variety of inputs, there are a total of 210 individual tests spanning 19 test suites.

The rest of this chapter goes over some of the finer points of the software design. Section 4.1 describes how the JIT is implemented without making concrete decision about the distributed memory model design. Section 4.2 describes how the code anticipates the implementation of path conditions on real routing tables. Section 4.3 describes the JITNIC configuration of standard JIT components.

## 4.1 Memory Abstraction

The implementation makes use of the advanced features of OCaml's module system in order make the system independent of the distributed memory model. Figure 4.1 gives the OCaml signature of the abstract memory model used throughout the implementation. The rest of the system is implemented as an OCaml functor, a module-level function that takes a concrete implementation of the memory signature and builds implementations of the small-step interpreter, symbolic evaluation, and test harness.

The current implementation of this signature used throughout the system is a trivial shared memory, in which both the CPU and the NIC may make unrestricted reads and writes to the same memory, whose representation type is a simple association list. This is clearly not a practical choice for the deployment of a system such as JITNIC, since a shared physical memory would lead to performance issues with both components. Instead, a more sophisticated implementation of distributed memory is needed, the details of which are an area of active research discussed in Chapter 6. Regardless, JITNIC's memory abstraction is expected to be general enough to describe a wide variety of more sophisticated memory models, and alternative implementations of the signature are under development.

## 4.2 Trace Identifiers

In Section 3.2, an important aspect of JITNIC was trivialized. The formalism allowed the keys of a table entry to be defined as an arbitrary logical predicate, which was

```
module type MemoryModel = sig
  type t
  val default : t
  val equal : t -> t -> bool
  val read : t -> value -> value
  val write : t -> value -> value -> t
end
```

Figure 4.1: The signature of the abstract memory model used in JITNIC's implementation. It includes an abstract datatype `t` and a default value representing an uninitialized memory configuration. An equality check is included for the purposes of testing, and the `read` and `write` are used to implement the interpreters and symbolic evaluator.

taken directly from the expression language. While this is convenient for the sake of the formalism, it raises concerns for implementation. While the current version of the JITNIC code uses the same convention as in Chapter 3, it also provides some convenient additional features that will help with the lift from theoretical to practical.

It is important to note that while the syntax of the formalism for table entries permits arbitrary logical formulas, the set of formulas actually installed by the symbolic evaluation phase is much smaller. A path condition $\varphi$ is generated as a sequentially constructed conjunction whose conjuncts are taken directly from the branch conditions in the program. Therefore, the complexity of the predicates is quite predictable, and it depends heavily on the choices for unary operations $\sim$ and binary operations $\odot$. Figure 4.2 gives JITNIC's OCaml definitions for these, a reasonable collection of operations curated with the eBPF instruction set in mind. The simple yet reasonably expressive selection of operators in the expression language allows JITNIC to build on work done for compilers of network programming languages. In particular, work has been done with boolean formulas consisting of conjunctions, disjunctions, and negations of atomic boolean formulas for the purpose of optimizing the compiler for NetKAT. The implementation of NetKAT's compiler is able to install table entries that implement boolean formulas of this form [Smolka et al., 2015]. Future versions of the JITNIC code will make use of this method to ensure that its generated path conditions can also be implemented on standard routing tables.

Despite the elegance of the existing solution, it is also the case that table entries implementing general boolean formulas of this form can be relatively expensive both in execution time and in table entry count. In an attempt to lighten this burden, the JITNIC

```
1  type uop =
2    | NEG (* integer negation *)
3    | NOT (* boolean negation *)
4
5  type bop =
6    | ADD (* integer arithmetic *)
7    | MUL
8    | EQ (* integer comparison operators *)
9    | NE
10   | LT
11   | LE
12   | GT
13   | GE
14   | LAND (* boolean [and] and [or] *)
15   | LOR
```

Figure 4.2

implementation introduces the notion of *trace identifiers*. A trace identifier is an integer value that identifies a unique trace from the original program. JITNIC implements a simple, low-overhead algorithm for generating these identifiers as part of the symbolic evaluation phase. First, it introduces an intermediate representation of actions called an annotated action and denoted with $a^+$. Annotated actions include the syntax of normal actions with the addition of annotations:

$$a^+ ::= \ldots \mid \mathbf{0} \mid \mathbf{1}$$

The symbolic evaluation phase outputs an annotated action, and instead of completely removing conditional branch commands from $c$, it leaves an annotation indicating which branch was actually taken – **0** for the `else` branch and **1** for the `if` branch. The ANN method defined in Figure 4.3 then pulls all the annotations out of the action after symbolic evaluation is complete, resulting in a string of bits that can be interpreted as a binary representation of an integer. Once the annotation-free action is mounted onto the pipeline using the MOUNT method from the previous chapter, fresh variables are assigned the integer values for the trace identifiers. This way, only the first table must provide the expensive implementation of the path conditions. The second table need only be concerned with the trace identifier, and the final table need only be concerned with the trace identifier in conjunction with a potential value read from memory.

$$\boxed{\textsc{Ann}(a^+)}$$

$$\begin{aligned}
\textsc{Ann}(\texttt{skip}) &\triangleq \langle \texttt{skip}, \langle\rangle \rangle \\
\textsc{Ann}(x := e) &\triangleq \langle x := e, \langle\rangle \rangle \\
\textsc{Ann}([e_1] := e_2) &\triangleq \langle [e_1] := e_2, \langle\rangle \rangle \\
\textsc{Ann}(x := [e]) &\triangleq \langle x := [e], \langle\rangle \rangle \\
\textsc{Ann}(\mathbf{0}) &\triangleq \langle \texttt{skip}, [0] \rangle \\
\textsc{Ann}(\mathbf{1}) &\triangleq \langle \texttt{skip}, [1] \rangle \\
\textsc{Ann}(a_1^+;\ a_2^+) &\triangleq \langle a_1;\ a_2, s_2 @ s_1 \rangle \\
&\quad \textit{where } \textsc{Ann}(a_1^+) = \langle a_1, s_1 \rangle \\
&\quad \textit{and } \textsc{Ann}(a_2^+) = \langle a_2, s_2 \rangle
\end{aligned}$$

Figure 4.3: The algorithm for extracting the trace identifier and corresponding action from an annotated action. The trace identifier gets extracted as a list of bits and then reinterpreted as an integer in binary.

## 4.3 JIT Configuration

As seen in Chapter 2, a tracing JIT must not only define a mapping from a source language to a target language, but also make design decisions about when to add and remove traces in the record. The part of JITNIC's design is interesting because of the dual role the CPU and NIC components serve. They operate both in the dynamic compiler paradigm, with the interpreter and monitor implemented on the CPU and the record implemented on the NIC, and in the network paradigm, with the CPU acting as the controller sending commands to the NIC, which implements routing policies on the tables. Because of this duality of purpose, it is effective to draw on design choices from routing tables and traffic engineering for the design of the JIT.

In the JITNIC implementation, a new trace is installed on the packet processing pipeline when it is first observed. When there is no more space on the routing tables for new entries, the least-recently used trace is evicted from the table. As to how many traces are installed on the pipeline at once, this depends entirely on the hardware. JITNIC's current implementation runs experiments on relatively small programs with a simulated routing table, so there are currently no explicit constraints on how many traces may be installed at once, and there is no explicit policy yet for evicting stale traces.

# Chapter 5

# Evaluation

In the last chapter, testing demonstrated the implementation's ability to perform correctly and at a sufficient level of generality to apply pre-compiled traces correctly to large sets of packets. However, this testing was done on small, impractical programs that were given two or three sets of inputs at most. In this section, I describe how the test harness was expanded to include programs that resemble computations one might actually deploy in a software defined network. Using the same prototype environment as before, I simulate JITNIC's execution of these programs on hundreds of packets. I not only check for correctness in the normal way but also evaluate the performance on important metrics such as the ratio of slow path executions to fast path executions and the peak number of traces installed on the pipeline.

While this method is unable to draw strong conclusions about the performance guarantees of JITNIC, it is able to approximate the conditions under which JITNIC provides statistically significant acceleration. In each of the three experiments, I consider the following inequality:

$$p_f \cdot c_f + p_s \cdot c_s < p \cdot c.$$

This compares the cost of running then entire program on the CPU to the cost of deploying JITNIC. The cost of a normal implementation is the the total number of packets $p$ times the cost $c$ of processing a single packet on a standard CPU implementation. Similarly, the cost of a JITNIC deployment is the sum of packets executed on the slow path and the packets executed on the fast path. Here, $p_f$ is the number of packets that get processed on the fast path at runtime, and $c_f$ is the average cost of processing a single packet on the fast path. Analogously, $p_s$ is the number of packets processed on the slow path at runtime and $c_s$ is cost of processing a packet on the slow path. If

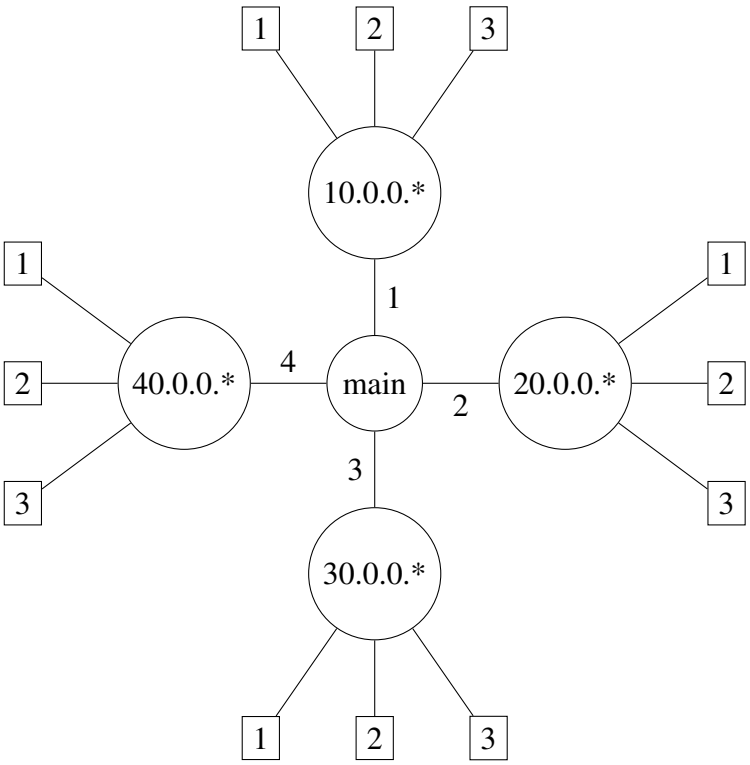


Figure 5.1: An example network topology motivating the first experiment. In a standard network configuration, the central switch is responsible for providing connectivity between end hosts in each of the four sub-networks, and would have a routing table that matches packet destinations to the proper ports on which to forward. For example, if a packet's destination is 10.0.0.2 range, it gets forwarded out port 1.

the inequality above holds for a given experiment, it is reasonable to conclude that it would be "worth it" to deploy JITNIC in the given scenario. Of course, the experiments are done in a prototype environment, so they produce values for $p$, $p_f$, and $p_s$ without attempting to measure exact values for $c$, $c_f$, and $c_s$. However, it is possible to make guesses about what they might be in relation to one another. Indeed, it is safe to expect $c_f$ is significantly smaller than $c$ because of how powerful state-of-the-art accelerators have become. It is also safe to assume that $c_s$ might be slightly larger than $c$, since both measure the speed of the same hardware component and $c_s$ measures an almost identical program with some additional runtime and memory overhead.

## 5.1 Basic Routing

In the first experiment, I design a program that is intended to implement a large routing table. An implementation of multiple table entries in the eBPF subset ought to have a

```
1 if in2 >= 10000 && in2 < 20000 then
2     out1 := 1; out2 := 1
3 else if in2 >= 20000 && in2 < 3000 then
4     out1 := 2; out2 := 1
5 else if in2 >= 30000 && in2 < 4000 then
6     out1 := 3; out2 := 1
7 else if in2 >= 40000 && in2 < 5000 then
8     out1 := 4; out2 := 1
9 else out1 := in1; out2 := 0
```

Figure 5.2: The command used in the first experiment. There are two inputs and two outputs: `in1` is the input port, `in2` is the destination address, `out1` is the output port, and `out2` indicates whether forwarding was successful. The program checks ranges of destination addresses and sets the outgoing port as would a standard routing table. If the destination address is unknown, the packet gets sent back to the source with a message indicating an unknown destination. The integer representations of the IP addresses in Figure 5.1 are simplified for the sake of readability.

large number of branches, and is therefore a good candidate for measuring the generality of the path conditions in a more practical setting. Figure 5.1 gives a motivating sample network topology, and Figure 5.2 gives the command used to implement the forwarding protocol for the central switch in this topology. Just like a routing table, the command checks the destination address of the packet and selects the next hop by setting the outgoing port for the packet. In this setting, however, the experiment simulates the performance of the central switch if JITNIC had been deployed to implement the packet-processing program, rather than an actual routing table.

I simulate and measure JITNIC's performance as $\sim 120$ packets from each of the 12 end hosts traverse the central switch. JITNIC's implementation is able to install a new table entry each time a new trace of the program is executed, subsequently producing correct output with 1435 out of 1440 packets ($\sim 99.7\%$) being executed on the NIC. This is done with a peak of 5 table entries per table, and the results persist across multiple executions when the ordering of input packets is randomized. With such a high ratio of fast-path packets to slow-path packets, and with such a management number of entries per table, a JITNIC implementation can be expected to provide a significant speedup in this scenario.

```
1 if in2 >= 10000 && in2 < 20000 then
2     out1 := 1; [in1 * 10 + out1] := in3; out2 := 1
3 else if in2 >= 20000 && in2 < 3000 then
4     out1 := 2; [in1 * 10 + out1] := in3; out2 := 1
5 else if in2 >= 30000 && in2 < 4000 then
6     out1 := 3; [in1 * 10 + out1] := in3; out2 := 1
7 else if in2 >= 40000 && in2 < 5000 then
8     out1 := 4; [in1 * 10 + out1] := in3 + 1; out2 := 1
9 else out1 := in1; [in1 * 10 + in1] := in3 + 1; out2 := 0
```

Figure 5.3: Similar to the previous experiment, this program implements a basic forwarding protocol. This time, however, the program get the packet size in `in3` and hashes it to a unique memory location. The hashing algorithm is of course quite brittle, but suffices for this example with only 4 ports while being simple enough to describe in the minimal eBPF subset language. Also, line 9 makes a write to memory in a semantically distinct manner from the rest of the branches, so that multiple traces may be installed on the second table.

## 5.2 Network Monitoring

While the previous experiment produces encouraging results, it fails to test the most challenging technical problem that informs JITNIC's design: memory. My next experiment measures JITNIC's performance on a program that makes frequent modifications to memory. This not only presents a challenge to the correctness of the software design, since memory consistency is difficult in a distributed settings, but also requires raises more difficult questions for the performance guarantees, as memory operations tend to introduce additional runtime overhead. For motivation, I draw on an important aspect of network implementation. Most network devices implement some form of traffic monitoring, be it a simple estimation of the volume of traffic or a more extensive log for exact values found in packet headers. Figure 5.3 gives a JITNIC command that implements a simplified version of a packet counter. Similar to the routing table in the previous section, it selects the outgoing port based on values in the packet header. This time however, it writes the packet size to a unique memory location based on the in port/out port pair, so that the device can track the number of bytes traversing each pair of ports.

Once again, I simulate and measure JITNIC's performance as $\sim 120$ packets from each of the 12 end hosts traverse the central switch and write their sizes to memory. Surprisingly, this produces identical results to the previous experiment. After closer

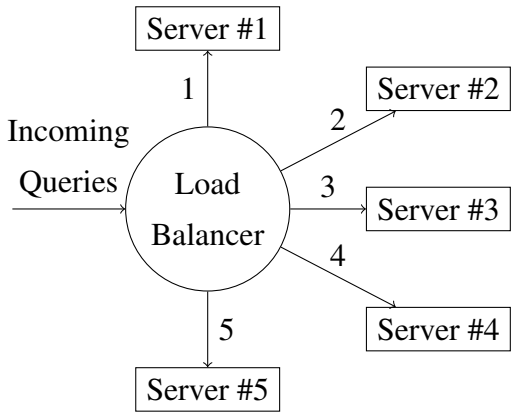


Figure 5.4: The topology of a load balancer and its supporting, identical servers.

examination, this is due to the fact that the monitoring program has the same number of traces as the routing program. Therefore, only 5 sets of table entries may be installed at most. The key difference is that the traces are now being installed across all three tables as a result of the write operands in the middle of the traces, and with various trace fragments installed on each table. Once again, such a high rate of fast-path packets is an encouraging sign for the potential deployment of a JITNIC design.

## 5.3 Load Balancing

The final experiment is intended to test the full extent of the expressive power of the JITNIC's implementation. Having rigorously tested the code on large, branching programs with writes to memory, I turn my attention to branching programs with reads, and in particular programs that branch on the values read from memory. This is significant because until this point packets have either been fully processed by the pipeline or fully processed by the CPU. Now, it will be possible for the pipeline to be partially applied to a given packet, read a value from memory that violates the path condition, discard the partial execution, and fall back to the slow path.

As a motivating example for this experiment, I turn to the standard network configuration paradigm or load balancing. Oftentimes in networking, the demand on a given end host resource is too high to be serviced by a single server. To remedy this, queries for the resource may first be routed to a load balancer, which is a device designed to distribute similar queries evenly across several identical servers providing the same service. There are many approaches to implementing a load balancer, but for the purposes of this experiment I will use a single counter that gets incremented after every packet,

```
1  x := [42];
2  if x % 5 = 0 then
3      out1 := 1
4  else if x % 5 = 1 then
5      out1 := 2
6  else if x % 5 = 2 then
7      out1 := 3
8  else if x % 5 = 3 then
9      out1 := 4
10 else out1 := 5
```

Figure 5.5: A program for reading the load balancer's counter from memory and selecting the appropriate port on which to forward the given query. Typically, the program would subsequently increment the counter, but due to the single-memory constraint this is done manually after each packet is processed.

and whose value is used by a modular computation to dynamically select an appropriate copy of the server (see Figures 5.4 and 5.5).[1]

After sending 100 query packets through this load balancing program, incrementing the value at memory address 42 after each packet, JITNIC's implementation of the program properly distributes the queries evenly among the 5 servers. As a result, 5 packets are processed on the CPU and 95 packets are processed on the NIC (95%), with a peak of 5 table entries per table. These results are made possible due to power of the JITNIC's formalism to express predicates on values read from memory combined with the ability of the packet-processing pipeline to check these predicates mid-execution and fall back to the slow path when needed.

[1] One might be concerned that the introduction of a modulo operator would create problems for the formalism since it has the side effect of throwing an exception on a divide-by-zero error. This can be avoided easily by requiring that the modulo operator only be applied with a non-zero constant as a right-hand operand, as is done in this program

# Chapter 6

# Related and Future Work

The JITNIC formalism and corresponding software system draw on research from the two fields of dynamic compilers and software-defined networking. This chapter briefly reviews some of the noteworthy contributions that have influenced JITNIC's design, before moving on to some of the immediate next steps and future applications of the system.

## 6.1 Related Work

The topic of dynamic compilers is a vibrant area in the field of programming language implementation. Most of the work to date has been related to popular high-level programming languages. For instance, tracing JITs have been used to optimize dynamically typed languages such as Python and JavaScript [Hackett and Guo, 2012, Gal et al., 2009] by identifying common typings of programs at runtime and compiling type-specialized traces. Methods from dynamic compilation have also been used to optimize garbage collection as an extension to the JavaScript VM [Clifford et al., 2015]. Moreover, JITs have been instrumental in the development of newer programming languages such as Julia, which is used for scientific computing [Bezanson et al., 2018].

JITNIC also builds on the related body of work in symbolic evaluation. The notion of partial or symbolic execution of a program was first introduced in the '70s as a means of software testing [King, 1976]. It has already been applied in the domain of network programming to test and debug P4 programs with Vera [Stoenescu et al., 2018]. The symbolic evaluation techniques used in the JITNIC formalism are based heavily on this approach, including the symbolic input variables and the symbolic variable environment.

## 6.2 Software-Defined Networking

Software-defined networking is also a young and exciting field of research on which JITNIC relies heavily. Of course, the design of new domain-specific languages such as eBPF and P4 is an important stepping-stone to the deployment of distributed JIT designs, and the long-term evolution of this new application will depend heavily on the direction taken by new devices and language features in networking. The immediate next step, however, will be to extend the JITNIC prototype environment into mapping from eBPF to P4. Since JITNIC is already implemented in OCaml, the Petr4 Framework [Doenges et al., 2021] is an excellent candidate for the target of the next version of JITNIC, and will open the door for the application of formal verification techniques to the full system.

Of course, formal methods and new DSLs are not the only side to SDN. CoNIC introduces a new technique for implementing distributed memory across the CPU and NIC called *treaties*. Instead of a shared memory or a cache implemented on the NIC, CoNIC proposes that the NIC maintain predicates over memory values under which packet-processing protocols remain valid. The CPU is responsible for notifying the NIC when these treaties become invalid, and thus stale table entries may be evicted from the NIC pipeline as memory values change. This is a convenient framework to pair with JITNIC, which produces the weakest preconditions under which packet headers and memory values reach different paths in the program. In fact, the new JIT design will allow for further study and more extensive empirical validation of new protocols such as CoNIC.

# Chapter 7

# Conclusions

This paper introduced JITNIC, a just-in-time compiler that provides a distributed implementation of packet-processing programs across the CPU and NIC to leverage the efficient, specialized network hardware. Traces from packet-processing programs running on the CPU are installed on the packet-processing pipeline, providing significant speed-up on most packets. The distributed approach was able to correctly describe the language semantics, and identify sufficiently general path conditions so that a large percentage of packets may be processed on the fast path. These properties were proved in a formal setting and validated in a software prototype by a collection of over 200 tests and 3 large-scale experiments.

In the future, the JITNIC implementation will be expanded to implement a mapping from the full source and target languages of eBPF and P4. This done, it will be possible to perform more rigorous experimentation by deploying the system on FPGAs. From there, JITNIC can be used to experiment with new implementations of distributed memory between the CPU and NIC, and the design template can be expanded to include other applications of partial in-network processing.

# Bibliography

[Apple, 2022] Apple (2022). Apple unveils M2, taking the breakthrough performance and capabilities of M1 even further. https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/. Press Release.

[Bezanson et al., 2018] Bezanson, J. et al. (2018). Julia: Dynamism and performance reconciled by design. *OOPSLA*.

[Cheng et al., 2011] Cheng, G. et al. (2011). Sketched oxide single-electron transistor. *Nature Nanotech*, 6:343–347.

[Clifford et al., 2015] Clifford, D. et al. (2015). Memento mori: dynamic allocation-site-based optimizations. *ISMM*.

[Doenges et al., 2021] Doenges, R. et al. (2021). Petr4: Formal foundations for p4 data planes. *POPL*.

[Gal et al., 2009] Gal, A. et al. (2009). Trace-based just-in-time type specialization for dynamic languages. *PLDI*.

[Hackett and Guo, 2012] Hackett, B. and Guo, S. (2012). Fast and precise hybrid type inference for JavaScript. *PLDI*.

[Hennessy and Patterson, 2019] Hennessy, J. L. and Patterson, D. A. (2019). A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60.

[Kelleher, 2022] Kelleher, A. (2022). Moore's law – now and in the future. https://www.intel.com/content/www/us/en/newsroom/opinion/moore-law-now-and-in-the-future.html.

[King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7).

[Moore, 1965] Moore, G. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[Moore, 1975] Moore, G. (1975). Progress in digital integrated electronics. *IEEE Technical Digest*, pages 11–13.

[Moore, 2015] Moore, G. (2015). Gordon Moore: The man whose name means progress, the visionary engineer reflects on 50 years of moore's law. `https://spectrum.ieee.org/gordon-moore-the-man-whose-name-means-progress`. Interviewed by Rachel Courtland.

[Smolka et al., 2015] Smolka, S. et al. (2015). A fast compiler for NetKAT. *ICFP*.

[Stoenescu et al., 2018] Stoenescu, R. et al. (2018). Debugging p4 programs with vera. *SIGCOMM*.

# Appendix A

# Semantics

## A.1 Definitions

$$n \in \mathbb{Z}$$

$$x \in \mathit{Var} \cup \{\mathtt{in}_1, \mathtt{in}_2, \ldots\} \cup \{\mathtt{out}_1, \mathtt{out}_2, \ldots\}$$

$$\begin{array}{rcl} \sim & ::= & - \mid \,! \\ \odot & ::= & \mathtt{+} \mid \mathtt{*} \mid \mathtt{=} \mid \mathtt{<>} \mid \mathtt{<} \mid \mathtt{<=} \mid \mathtt{>} \mid \mathtt{>=} \mid \mathtt{\&\&} \mid \mathtt{||} \end{array}$$

$$\begin{array}{rcl} e & ::= & n \\ & \mid & x \\ & \mid & \sim e \\ & \mid & e_1 \odot e_2 \end{array} \qquad \begin{array}{rcl} a & ::= & \mathtt{skip} \\ & \mid & x := e \\ & \mid & x := [e] \\ & \mid & [e_1] := e_2 \\ & \mid & a_1; a_2 \end{array} \qquad \begin{array}{rcl} c & ::= & \mathtt{skip} \\ & \mid & x := e \\ & \mid & x := [e] \\ & \mid & [e_1] := e_2 \\ & \mid & c_1; c_2 \\ & \mid & \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \end{array}$$

$$\sigma : \mathbb{Z} \rightharpoonup \mathbb{Z}$$

$$\varepsilon : \mathit{Var} \rightharpoonup \mathbb{Z}$$

For the convenience of defining inference rules, I adopt the following abuse of notation:

$$\sigma(n) = \begin{cases} \sigma(n) & \text{if } n \in \mathit{dom}(\sigma) \\ 0 & \text{otherwise} \end{cases}$$

This is also done for the variable assignment $\varepsilon$ and later with the symbolic variable environment $\mathcal{E}$ (see next chapter).

**Definition A.1.** *Two commands $c_1$ and $c_2$ [actions $a_1$ and $a_2$] are said to be semantically equivalent if for every initial memory configuration $\sigma$ and variable assignment $\varepsilon$ they multi-step (see below) to the same final configuration i.e. there exist some $\sigma'$ and some $\varepsilon'$ such that $\langle \sigma, \varepsilon, c_1[a_1] \rangle \to^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$ and $\langle \sigma, \varepsilon, c_2[a_2] \rangle \to^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$.*

## A.2 Small-Step Semantics

$\boxed{\langle \varepsilon, e \rangle \Downarrow n}$

$$\frac{\varepsilon(x) = n}{\langle \varepsilon, x \rangle \Downarrow n}\ \text{E-VAR} \qquad \frac{\langle \varepsilon, e \rangle \Downarrow n' \quad \sim n' = n}{\langle \varepsilon, \sim e \rangle \Downarrow n}\ \text{E-UOP}$$

$$\frac{\langle \varepsilon, e_1 \rangle \Downarrow n_1 \quad \langle \varepsilon, e_2 \rangle \Downarrow n_2 \quad n_1 \odot n_2 = n}{\langle \varepsilon, e_1 \odot e_2 \rangle \Downarrow n}\ \text{E-BOP}$$

$\boxed{\langle \sigma, \varepsilon, a \rangle \to \langle \sigma', \varepsilon', a' \rangle}$

$$\frac{\varepsilon' = \varepsilon[x \mapsto n] \quad \langle \varepsilon, e \rangle \Downarrow n}{\langle \sigma, \varepsilon, x := e \rangle \to \langle \sigma, \varepsilon', \texttt{skip} \rangle}\ \text{A-ASGN} \qquad \frac{\varepsilon' = \varepsilon[x \mapsto \sigma(n)] \quad \langle \varepsilon, e \rangle \Downarrow n}{\langle \sigma, \varepsilon, x := [e] \rangle \to \langle \sigma, \varepsilon', \texttt{skip} \rangle}\ \text{A-LD}$$

$$\frac{\sigma' = \sigma[n_1 \mapsto n_2] \quad \langle \varepsilon, e_1 \rangle \Downarrow n_1 \quad \langle \varepsilon, e_2 \rangle \Downarrow n_2}{\langle \sigma, \varepsilon, [e_1] := e_2 \rangle \to \langle \sigma', \varepsilon, \texttt{skip} \rangle}\ \text{A-ST}$$

$$\frac{}{\langle \sigma, \varepsilon, \texttt{skip}; a \rangle \to \langle \sigma, \varepsilon, a \rangle}\ \text{A-SEQ-CONT}$$

$$\frac{\langle \sigma, \varepsilon, a_1 \rangle \to \langle \sigma', \varepsilon', a_1' \rangle}{\langle \sigma, \varepsilon, a_1; a_2 \rangle \to \langle \sigma', \varepsilon', a_1'; a_2 \rangle}\ \text{A-SEQ-STEP}$$

$\boxed{\langle \sigma, \varepsilon, c\rangle \to \langle \sigma', \varepsilon', c'\rangle}$

$$\frac{\varepsilon' = \varepsilon[x \mapsto n] \quad \langle \varepsilon, e\rangle \Downarrow n}{\langle \sigma, \varepsilon, x := e\rangle \to \langle \sigma, \varepsilon', \texttt{skip}\rangle}\ \text{C-ASGN} \qquad \frac{\varepsilon' = \varepsilon[x \mapsto \sigma(n)] \quad \langle \varepsilon, e\rangle \Downarrow n}{\langle \sigma, \varepsilon, x := [e]\rangle \to \langle \sigma, \varepsilon', \texttt{skip}\rangle}\ \text{C-LD}$$

$$\frac{\sigma' = \sigma[n_1 \mapsto n_2] \quad \langle \varepsilon, e_1\rangle \Downarrow n_1 \quad \langle \varepsilon, e_2\rangle \Downarrow n_2}{\langle \sigma, \varepsilon, [e_1] := e_2\rangle \to \langle \sigma', \varepsilon, \texttt{skip}\rangle}\ \text{C-ST}$$

$$\frac{}{\langle \sigma, \varepsilon, \texttt{skip}; c\rangle \to \langle \sigma, \varepsilon, c\rangle}\ \text{C-SEQ-CONT}$$

$$\frac{\langle \sigma, \varepsilon, c_1\rangle \to \langle \sigma', \varepsilon', c_1'\rangle}{\langle \sigma, \varepsilon, c_1; c_2\rangle \to \langle \sigma', \varepsilon', c_1'; c_2\rangle}\ \text{C-SEQ-STEP}$$

$$\frac{\langle \varepsilon, e\rangle \Downarrow n \quad n \neq 0}{\langle \sigma, \varepsilon, \texttt{if } e \texttt{ then } c_t \texttt{ else } c_f\rangle \to \langle \sigma, \varepsilon, c_t\rangle}\ \text{C-BR-T}$$

$$\frac{\langle \varepsilon, e\rangle \Downarrow 0}{\langle \sigma, \varepsilon, \texttt{if } e \texttt{ then } c_t \texttt{ else } c_f\rangle \to \langle \sigma, \varepsilon, c_f\rangle}\ \text{C-BR-F}$$

$\boxed{\langle \sigma, \varepsilon, a\rangle \to^* \langle \sigma', \varepsilon', a'\rangle}$

$$\frac{}{\langle \sigma, \varepsilon, a\rangle \to^* \langle \sigma, \varepsilon, a\rangle}\ \text{A-REFL}$$

$$\frac{\langle \sigma, \varepsilon, a\rangle \to \langle \sigma'', \varepsilon'', a''\rangle \quad \langle \sigma'', \varepsilon'', a''\rangle \to^* \langle \sigma', \varepsilon', a'\rangle}{\langle \sigma, \varepsilon, a\rangle \to^* \langle \sigma', \varepsilon', a'\rangle}\ \text{A-TRANS}$$

$\boxed{\langle \sigma, \varepsilon, c\rangle \to^* \langle \sigma', \varepsilon', c'\rangle}$

$$\frac{}{\langle \sigma, \varepsilon, c\rangle \to^* \langle \sigma, \varepsilon, c\rangle}\ \text{C-REFL}$$

$$\frac{\langle \sigma, \varepsilon, c\rangle \to \langle \sigma'', \varepsilon'', c''\rangle \quad \langle \sigma'', \varepsilon'', c''\rangle \to^* \langle \sigma', \varepsilon', c'\rangle}{\langle \sigma, \varepsilon, c\rangle \to^* \langle \sigma', \varepsilon', c'\rangle}\ \text{C-TRANS}$$

## A.3 Results

**Theorem A.1 (Termination of Expressions).** *For any expression e under any variable assignment* $\varepsilon$*, there exists an integer n s.t.* $\langle \varepsilon, e\rangle \Downarrow n$.

*Proof.* Trivial, using induction on the structure of $e$.

□

**Theorem A.2 (Determinism of Expressions).** *For any expression e under and variable assignment* $\varepsilon$*, and for any integers n and* $n'$*, if* $\langle \varepsilon, e \rangle \Downarrow n$ *and* $\langle \varepsilon, e \rangle \Downarrow n'$ *then* $n = n'$*.*

*Proof.* Trivial, using induction on the structure of $e$.

□

**Theorem A.3 (Termination of Commands [Actions]).** *For any command c [action a], any initial memory configuration* $\sigma$*, and any initial variable assignment* $\varepsilon$*, there exist* $\sigma'$*, and* $\varepsilon'$ *such that* $\langle \sigma, \varepsilon, c[a] \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$.

*Proof.* I proceed by induction on the structure of $c$ $[a]$. The base cases are all trivial, since for `skip` the claim holds by reflexivity and all the other base cases (assignment, load, and store) take a single step to a configuration satisfying the termination claim.

For the inductive case, $c = c_1; c_2$ $[a = a_1; a_2]$, the inductive hypothesis ensures, given a $\sigma$ and $\varepsilon$, both that $\langle \sigma, \varepsilon c_1[a_1] \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip}$ for some fixed $\sigma'$ and $\varepsilon'$ and that $\langle \sigma', \varepsilon', c_2[a_2] \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$ for the same $\sigma'$ and $\varepsilon'$ and new fixed $\sigma''$ and $\varepsilon''$. Therefore, these sequences of steps may be chained together using a single application of C-SEQ-CONT [C-SEQ-CONT] as follows:

$$\langle \sigma, \varepsilon, c_1; c_2[a_1; a_2] \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip}; c_2[\texttt{skip}; a_2] \rangle \rightarrow \langle \sigma', \varepsilon', c_2[a_2] \rangle \rightarrow^* \langle \sigma'', \varepsilon'', \texttt{skip} \rangle$$

[This concludes the proof of termination for actions.]

For the other inductive case, let $c = \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f$. Either $e$ is true or false in the variable assignment, and in either case the command takes a single step to $c_t$ or $c_f$ both of which multi-step to a final configuration by the inductive hypothesis.

□

**Theorem A.4.** *For any commands c, c′, and c″ [any actions a, a′, and a″], any* $\sigma$*,* $\sigma'$*, and* $\sigma''$*, and any* $\varepsilon$*,* $\varepsilon'$*, and* $\varepsilon''$*, if* $\langle \sigma, \varepsilon, c[a] \rangle \rightarrow \langle \sigma', \varepsilon', c'[a'] \rangle$ *and* $\langle \sigma, \varepsilon, c[a] \rangle \rightarrow \langle \sigma'', \varepsilon'', c''[a''] \rangle$*, then* $\sigma' = \sigma''$*,* $\varepsilon' = \varepsilon''$*, and* $c' = c''[a' = a'']$*.*

*Proof.* The proof is done by case analysis on the structure of $c$ $[a]$. All of the base cases follow trivially from the determinism of expression evaluation, since there is only one rule allowing each of them to step. If $c = c_1; c_2$ $[a = a_1; a_2]$ the next single step is uniquely determined by whether $a_1 = \texttt{skip}$. [This concludes the proof for actions.] Finally, if $c$ is a conditional, then the next single step is uniquely determined by the value the conditional guard expression $e$, and the claim follows from the determinism of expression evaluation.

□

**Theorem A.5 (Determinism of Commands [Actions]).** *For any command c [action a], any* $\sigma$*,* $\sigma'$*, and* $\sigma''$*, and any* $\varepsilon$*,* $\varepsilon'$*, and* $\varepsilon''$*, if* $\langle \sigma, \varepsilon, c[a] \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$ *and* $\langle \sigma, \varepsilon, c[a] \rangle \rightarrow^* \langle \sigma'', \varepsilon'', \texttt{skip} \rangle$*, then* $\sigma' = \sigma''$ *and* $\varepsilon' = \varepsilon''$*.*

*Proof.* I proceed by induction on the structure of the proof tree for the multi-step relation. The base case C-REFL [A-REFL] is trivial, and the inductive case C-TRANS [A-TRANS] follows from the previous theorem regarding the single-step relation and the inductive hypothesis.

□

**Theorem A.6 (Idempotence of `Skip`).** *Any action a is semantically equivalent to both* `skip`$;a$ *and* $a;$ `skip`*.*

*Proof.* Both equivalences hold as a consequence of termination and determinism of actions. The action `skip`$;a$ takes a single step to $a$ under and memory configuration and variable assignment. Similarly, $a;$`skip` multi-steps to `skip;skip` making the same modifications to the memory and variable assignment as the multi-step for $a$ before taking a final, trivial step to termination.

□

**Theorem A.7 (Associativity of Sequencing).** *For any three actions* $a_1$*,* $a_2$*, and* $a_3$*, it is the case that* $(a_1;a_2);a_3$ *is semantically equivalent to* $a_1;(a_2;a_3)$*.*

*Proof.* This is another consequence of termination and determinism. Regardless of left or right associativity, both actions step $a_1$ to termination. For left-associative sequencing, the result is $(\texttt{skip};a_2);a_3$, whereas for right-associative sequencing the result is $\texttt{skip};(a_2;a_3)$, both of which take a single step to $a_2;a_3$.

□

# Appendix B

# Symbolic Evaluation

## B.1 Definitions

$$b \in \mathbb{B}$$

$$\varphi \in e$$

Symbolic values $\alpha_n$ are left abstract, but correspond to the input variables $\texttt{in}_n$ by convention. Consequently, the notation for concrete big-step evaluation of expressions $\langle \varepsilon, e \rangle \Downarrow n$ will also be defined for symbolic values $\langle \varepsilon, \nu \rangle \Downarrow n$ where each $\alpha_n$ is understood to refer to the input variable $\texttt{in}_n$.

$$\nu ::= n \mid \alpha \mid \sim \nu \mid \nu_1 \odot \nu_2$$

$$\mathcal{E} : \mathit{Var} \rightharpoonup \nu$$

**Definition B.1.** *A variable assignment* $\varepsilon$ *satisfies a condition* $\varphi$*, denoted* $\varphi(\varepsilon)$*, iff.* $\langle \varepsilon, \varphi \rangle \Downarrow n$ *and* $n \neq 0$*.*

**Definition B.2.** *A symbolic environment* $\mathcal{E}$ *is considered consistent with a variable assignment* $\varepsilon$ *if (1) dom* $\varepsilon = dom\ \mathcal{E}$*, (2) for every* $\texttt{in}_n \in dom\ \varepsilon$*,* $\mathcal{E}(\texttt{in}_n) = \alpha_n$*, and (3) for all the other variables x in the shared domain,* $\langle \varepsilon, \mathcal{E}(x) \rangle \Downarrow \varepsilon(x)$*.*

## B.2 Small-Step Symbolic Evaluation

$\boxed{\langle \mathcal{E}, e\rangle \downarrow \nu}$

$$\frac{\mathcal{E}(x) = \nu}{\langle \mathcal{E}, x\rangle \downarrow \nu}\ \text{SE-VAR} \qquad \frac{\langle \mathcal{E}, e\rangle \downarrow n' \quad \sim n' = n}{\langle \mathcal{E}, \sim e\rangle \downarrow n}\ \text{SE-UOP-CON}$$

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu}{\langle \mathcal{E}, \sim e\rangle \downarrow \sim \nu}\ \text{SE-UOP-ABS}$$

$$\frac{\langle \mathcal{E}, e_1\rangle \downarrow n_1 \quad \langle \mathcal{E}, e_2\rangle \downarrow n_2 \quad n_1 \odot n_2 = n}{\langle \mathcal{E}, e_1 \odot e_2\rangle \downarrow n}\ \text{SE-BOP-CON}$$

$$\frac{\langle \mathcal{E}, e_1\rangle \downarrow \nu_1 \quad \langle \mathcal{E}, e_2\rangle \downarrow \nu_2}{\langle \mathcal{E}, e_1 \odot e_2\rangle \downarrow \nu_1 \odot \nu_2}\ \text{SE-BOP-ABS}$$

$\boxed{\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma', \varepsilon', c', a', \varphi_1', \varphi_2', b', \mathcal{E}'\rangle}$

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \mathcal{E}' = \mathcal{E}[x \mapsto \nu]}{\langle \sigma, \varepsilon, x := e, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, \texttt{skip}, a; x := \nu, \varphi_1, \varphi_2, b, \mathcal{E}'\rangle}\ \text{SC-ASGN}$$

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \langle \varepsilon, \nu\rangle \Downarrow n \quad \varepsilon' = \varepsilon[\texttt{in}_0 \mapsto \sigma(n)] \quad \mathcal{E}' = \mathcal{E}[x \mapsto \alpha_0] \quad \texttt{in}_0 \textit{ is fresh}}{\langle \sigma, \varepsilon, x := [e], a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon', \texttt{skip}, a; x := [\nu], \varphi_1, \varphi_2, \texttt{true}, \mathcal{E}'\rangle}\ \text{SC-LD}$$

$$\frac{\langle \mathcal{E}, e_1\rangle \downarrow \nu_1 \quad \langle \mathcal{E}, e_2\rangle \downarrow \nu_2 \quad \langle \varepsilon, \nu_1\rangle \Downarrow n_1 \quad \langle \varepsilon, \nu_2\rangle \Downarrow n_2 \quad \sigma' = \sigma[n_1 \mapsto n_2]}{\langle \sigma, \varepsilon, [e_1] := e_2, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma', \varepsilon, \texttt{skip}, a; [\nu_1] := \nu_2, \varphi_1, \varphi_2, \texttt{true}, \mathcal{E}\rangle}\ \text{SC-ST}$$

$$\frac{}{\langle \sigma, \varepsilon, \texttt{skip}; c, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle}\ \text{SC-SEQ-CONT}$$

$$\frac{\langle \sigma, \varepsilon, c_1, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma', \varepsilon', c_1', a', \varphi_1', \varphi_2', b', \mathcal{E}'\rangle}{\langle \sigma, \varepsilon, c_1; c_2, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma', \varepsilon', c_1'; c_2, a', \varphi_1', \varphi_2', b', \mathcal{E}'\rangle}\ \text{SC-SEQ-STEP}$$

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \langle \varepsilon, \nu\rangle \Downarrow n \quad n \neq 0 \quad \varphi_1' = \nu\ \texttt{\&\&}\ \varphi_1}{\langle \sigma, \varepsilon, \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f, a, \varphi_1, \varphi_2, \texttt{false}, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, c_t, a, \varphi_1', \varphi_2, \texttt{false}, \mathcal{E}\rangle}\ \text{SC-BR-T1}$$

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \langle \varepsilon, \nu\rangle \Downarrow n \quad n \neq 0 \quad \varphi_2' = \nu\ \texttt{\&\&}\ \varphi_2}{\langle \sigma, \varepsilon, \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f, a, \varphi_1, \varphi_2, \texttt{true}, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, c_t, a, \varphi_1, \varphi_2', \texttt{true}, \mathcal{E}\rangle}\ \text{SC-BR-T2}$$

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \langle \varepsilon, \nu\rangle \Downarrow 0 \quad \varphi_1' = !\nu\ \texttt{\&\&}\ \varphi_1}{\langle \sigma, \varepsilon, \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f, a, \varphi_1, \varphi_2, \texttt{false}, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, c_f, a, \varphi_1', \varphi_2, \texttt{false}, \mathcal{E}\rangle}\ \text{SC-BR-F1}$$

$$\frac{\langle \mathcal{E}, e\rangle \downarrow \nu \quad \langle \varepsilon, \nu\rangle \Downarrow 0 \quad \varphi_2' = !\nu\ \texttt{\&\&}\ \varphi_2}{\langle \sigma, \varepsilon, \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f, a, \varphi_1, \varphi_2, \texttt{true}, \mathcal{E}\rangle \Rightarrow \langle \sigma, \varepsilon, c_f, a, \varphi_1, \varphi_2', \texttt{true}, \mathcal{E}\rangle}\ \text{SC-BR-F2}$$

$\boxed{\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle \Rightarrow^* \langle \sigma', \varepsilon', c', a', \varphi_1', \varphi_2', b', \mathcal{E}' \rangle}$

$$\frac{}{\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle \Rightarrow^* \langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle}\ \text{SC-REFL}$$

$$\frac{\begin{array}{c}\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle \Rightarrow \langle \sigma'', \varepsilon'', c'', a'', \varphi_1'', \varphi_2'', b'', \mathcal{E}'' \rangle \\ \langle \sigma'', \varepsilon'', c'', a'', \varphi_1'', \varphi_2'', b'', \mathcal{E}'' \rangle \Rightarrow^* \langle \sigma', \varepsilon', c', a', \varphi_1', \varphi_2', b', \mathcal{E}' \rangle\end{array}}{\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle \Rightarrow^* \langle \sigma', \varepsilon', c', a', \varphi_1', \varphi_2', b', \mathcal{E}' \rangle}\ \text{SC-TRANS}$$

## B.3 Results

**Theorem B.1 (Determinism of Symbolic Evaluation of Expressions).** *For any expression $e$ under any variable assignment $\varepsilon$ and for any symbolic values $\nu$ and $\nu'$, if $\langle \varepsilon, e \rangle \downarrow \nu$ and $\langle \varepsilon, e \rangle \downarrow \nu'$, then $\nu = \nu'$*

*Proof.* Trivial, by induction on the structure of $e$.

□

**Theorem B.2 (Semantics Preservation of Expressions).** *For all expressions $e$, all variable assignments $\varepsilon$, and all symbolic environments $\mathcal{E}$ consistent with $\varepsilon$, if $\langle \mathcal{E}, e \rangle \downarrow \nu$, if $\langle \varepsilon, \nu \rangle \Downarrow n$, and if $\langle \varepsilon, e \rangle \Downarrow n'$, then $n = n'$.*

*Proof.* The proof is by induction on the structure of $e$. The case $e = n$ is trivial. In the variable case, the claim follows from consistency. Both inductive cases follow from the inductive hypothesis.

□

**Theorem B.3 (Semantics Preservation of Commands).** *For any commands $c$, any initial memory configuration $\sigma$, any variable assignments $\varepsilon$, any action $a$, any predicates $\varphi_1$ and $\varphi_2$, any indicator $b$, and any symbolic environment $\mathcal{E}$ consistent with $\varepsilon$, if*

$$\langle \sigma, \varepsilon, a \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle,$$

*and if*

$$\langle \sigma', \varepsilon', c, a, \varphi_1, \varphi_1, b, \mathcal{E} \rangle \Rightarrow^* \langle \sigma_1, \varepsilon_1, \texttt{skip}, a', \varphi_1', \varphi_2', b', \mathcal{E}_1 \rangle,$$

*and if*

$$\langle \sigma, \varepsilon, a' \rangle \rightarrow^* \langle \sigma_2, \varepsilon_2, \texttt{skip} \rangle, \text{ and } \langle \sigma, \varepsilon, a; c \rangle \rightarrow^* \langle \sigma_3, \varepsilon_3, \texttt{skip} \rangle,$$

*then* $\sigma_2 = \sigma_3$ *and* $\varepsilon_2 = \varepsilon_3$. [1]

*Proof.* I proceed by induction on the structure of $c$. The `skip` case is trivial, and the rest of the base cases follow from semantics preservation of expression evaluation. The sequencing case follows from the inductive hypothesis, and the conditional case follows from the inductive hypothesis along with the observation that both judgements $\rightarrow$ and $\Rightarrow$ have the same semantics for taking a single step on a conditional.

□

**Theorem 3.1 (Semantics Preservation).** *Given a memory configuration* $\sigma$*, an* $\varepsilon$ *that is defined only on the input variables, and a command c. Let* $\mathcal{E}$ *be the initial symbolic environment defined only on the input variables, so that* $\mathcal{E}(\texttt{in}_n) = \alpha_n$ *for all* $\texttt{in}_n$ *defined in* $\varepsilon$*. Then if* $\langle \sigma, \varepsilon, c\rangle \rightarrow^* \langle \sigma_1, \varepsilon_1, \texttt{skip}\rangle$*, if* $\langle \sigma, \varepsilon, c, \texttt{skip}, 1, 1, \texttt{false}, \mathcal{E}\rangle \Rightarrow^* \langle \sigma_2, \varepsilon_2, \texttt{skip}, a, \varphi_1, \varphi_2, b, \mathcal{E}'\rangle$*, and if* $\langle \sigma, \varepsilon, a\rangle \rightarrow^* \langle \sigma_3, \varepsilon_3, \texttt{skip}\rangle$*, then* $\sigma_2 = \sigma_3$ *and* $\varepsilon_1$ *agrees with* $\varepsilon_3$ *on the output variables.*

*Proof.* This is a special case of theorem B.2.

□

**Theorem B.3.** *Given a memory configuration* $\sigma$*, a variable assignment* $\varepsilon$*, a command c, an action a, partial path conditions* $\varphi_1$ *and* $\varphi_2$*, an indicator b, and a symbolic environment* $\mathcal{E}$ *consistent with* $\varepsilon$*, and let*

$$\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma', \varepsilon', c', a', \varphi_1', \varphi_2', b', \mathcal{E}'\rangle.$$

*Then for any* $\varepsilon_0$ *satisfying* $\varphi = \varphi_1 \& \varphi_2$*, if*

$$\langle \sigma, \varepsilon_0, c, a, \varphi_1, \varphi_2, b, \mathcal{E}\rangle \Rightarrow \langle \sigma'', \varepsilon'', c'', a'', \varphi_1'', \varphi_2'', b'', \mathcal{E}''\rangle.,$$

*then* $a' = a''$*,* $\varphi_1' = \varphi_1''$*, and* $\varphi_2' = \varphi_2''$*.*

*Proof.* The proof is by case analysis on the structure of $c$. All cases except the conditional case are trivial, since they do not modify the path conditions and $a' = a''$ follows by the determinism of symbolic evaluation of expressions. In the case of conditionals, there are four different cases based on the evaluation of the guard $e$ and the value for $b$, but the proof is analogous in each. I simply observe that the update to $\varphi_1$ or $\varphi_2$ is

[1] The sequencing of $a$ and $c$ here is an abuse of notation. However, since the cases for actions are just a subset of the cases for commands, a trivial mapping from actions to commands exists which allows actions to be lifted into the context of commands.

uniquely determined by the symbolic evaluation of $e$, and that this is unaffected by differences between $\varepsilon_0$ and $\varepsilon$.

□

**Theorem B.4.** *Given a memory configuration $\sigma$, a variable assignment $\varepsilon$, a command $c$, an action $a$, partial path conditions $\varphi_1$ and $\varphi_2$, an indicator $b$, and a symbolic environment $\mathcal{E}$ consistent with $\varepsilon$, and let*

$$\langle \sigma, \varepsilon, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle \Rightarrow^* \langle \sigma', \varepsilon', c', a', \varphi_1', \varphi_2', b', \mathcal{E}' \rangle.$$

*Then for any $\varepsilon_0$ satisfying $\varphi = \varphi_1 \& \varphi_2$, if*

$$\langle \sigma, \varepsilon_0, c, a, \varphi_1, \varphi_2, b, \mathcal{E} \rangle \Rightarrow^* \langle \sigma'', \varepsilon'', c'', a'', \varphi_1'', \varphi_2'', b'', \mathcal{E}'' \rangle.,$$

*then $a' = a''$, $\varphi_1' = \varphi_1''$, and $\varphi_2' = \varphi_2''$.*

*Proof.* The proof is by induction on the structure of the proof tree for the multi-step relation $\rightarrow^*$. The base case SE-REFL is trivial, and the inductive case SE-TRANS follows from the previous theorem and the inductive hypothesis.

□

**Theorem 3.2 (Precondition Soundness).** *Given a memory store $\sigma$, an $\varepsilon$ defined only on the input variables, and a command $c$. Let $\mathcal{E}$ be defined as in Theorem 3.1, and let $\langle \sigma, \varepsilon, c, \texttt{skip}, 1, 1, \texttt{false}, \mathcal{E} \rangle \Rightarrow^* \langle \sigma_1, \varepsilon', \texttt{skip}, a, \varphi_1, \varphi_2, b, \mathcal{E}_1 \rangle$. Then for any $\varepsilon_0$ that satisfies $\varphi = \varphi_1 \,\&\, \varphi_2$, if $\langle \sigma, \varepsilon_0, c, \texttt{skip}, 1, 1, \texttt{true}, \mathcal{E} \rangle \Rightarrow^* \langle \sigma_2, \varepsilon_0', \texttt{skip}, a', \varphi_1', \varphi_2', b, \mathcal{E}_2 \rangle$, then we have $a = a'$, $\varphi_1 = \varphi_1'$, and $\varphi_2 = \varphi_2'$.*

*Proof.* This is a special case of the previous theorem.

□

**Theorem 3.3.** *If $\text{MOUNT}(a) = \langle a_1, a_2, a_3 \rangle$, then $a_1$ and $a_3$ contain no memory operation while $a_2$ consists of a single load or store, or is `skip`. Moreover, $a$ is semantically equivalent to $a_1$; $a_2$; $a_3$, that is to say, $\langle \sigma, \varepsilon, a \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$ iff. $\langle \sigma, \varepsilon, a_1;\ a_2;\ a_3 \rangle \rightarrow^* \langle \sigma', \varepsilon', \texttt{skip} \rangle$.*

*Proof.* I proceed by induction on the structure of $a$, relying on results in the previous chapter regarding the idempotence of `skip` and the associativity of sequencing.

The base cases, where $a = \texttt{skip}$, $a = x := e$, $a = x := [e]$, and $a = [e_1] := e_2$ are all trivial. Each base case of the MOUNT function satisfies the conditions trivially, and the outputs are all semantically equivalent to the inputs by the idempotence of `skip`.

For the inductive case, where $a = a_1;a_2$, it follows from the inductive hypothesis that both $\textsc{Mount}(a_1) = \langle a_{1a}, a_{1b}, a_{1c}\rangle$ and $\textsc{Mount}(a_2) = \langle a_{2a}, a_{2b}, a_{2c}\rangle$ satisfy the structural conditions. Since this is all done under the assumption that $a$ only includes one memory operation, either (1) $a_{1b}$ is a single memory operation, (2) $a_{2b}$ is a single memory operation, or (3) there is no memory operation, and there are no other memory operations. In case of (1), $\textsc{Mount}(a) = \langle a_{1a}, a_{1b}, a_{1c}; a_2\rangle$, satisfying the structural condition. Likewise for (2), $\textsc{Mount}(a) = \langle a_1; a_{2a}, a_{2b}, a_{2c}\rangle$, also satisfying the structural condition. In both cases, semantic equality holds by the inductive hypothesis and by the associativity of sequencing. Finally, if (3) holds then the structural conditions are satisfied trivially and semantic equality holds by the idempotence of `skip`.

□

# Appendix C

# Case Study

This chapter gives a brief example of how one might modify the symbolic evaluation rules given in the previous chapter for a different NIC architecture. Instead of the previous case in which the pipeline consists of one memory unit and pre- and post-processing units, this case assumes and architecture that can handle arbitrary alternating memory and processing units. The notion of a stack of predicates is introduced, followed by a sample collection of alternative symbolic evaluation rules:

$s ::= \langle\rangle \mid \varphi :: s$

$$\boxed{\langle \sigma, \varepsilon, c, a, s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma', \varepsilon', c', a', s', n', \mathcal{E}' \rangle}$$

$$\frac{\langle \mathcal{E}, e \rangle \downarrow \nu \quad \mathcal{E}' = \mathcal{E}[x \mapsto \nu]}{\langle \sigma, \varepsilon, x := e, a, s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma, \varepsilon, \texttt{skip}, a; x := \nu, s, n, \mathcal{E}' \rangle} \text{ SC}^+\text{-ASGN}$$

$$\frac{\langle \mathcal{E}, e \rangle \downarrow \nu \quad \langle \varepsilon, \nu \rangle \Downarrow n \quad \varepsilon' = \varepsilon[\texttt{in}_n \mapsto n] \quad \mathcal{E}' = \mathcal{E}[x \mapsto \alpha_n] \quad \texttt{in}_n \textit{ is fresh}}{\langle \sigma, \varepsilon, x := [e], a, s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma, \varepsilon', \texttt{skip}, a; x := [\nu], 1 :: s, n+1, \mathcal{E}' \rangle} \text{ SC}^+\text{-LD}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \downarrow \nu_1 \quad \langle \mathcal{E}, e_2 \rangle \downarrow \nu_2 \quad \langle \varepsilon, \nu_1 \rangle \Downarrow n_1 \quad \langle \varepsilon, \nu_1 \rangle \Downarrow n_2 \quad \sigma' = \sigma[n_1 \mapsto n_2]}{\langle \sigma, \varepsilon, [e_1] := e_2, a, s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma', \varepsilon, \texttt{skip}, a; [\nu_1] := \nu_2, 1 :: s, n+1, \mathcal{E} \rangle} \text{ SC}^+\text{-ST}$$

$$\frac{}{\langle \sigma, \varepsilon, \texttt{skip}; c, a, s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma, \varepsilon, c, a, s, n, \mathcal{E} \rangle} \text{ SC}^+\text{-SEQ-CONT}$$

$$\frac{\langle \sigma, \varepsilon, c_1, a, s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma', \varepsilon', c_1', a', s', n', \mathcal{E}' \rangle}{\langle \sigma, \varepsilon, c_1; c_2, a, s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma', \varepsilon', c_1'; c_2, a', s', n', \mathcal{E}' \rangle} \text{ SC}^+\text{-SEQ-STEP}$$

$$\frac{\langle \mathcal{E}, e \rangle \downarrow \nu \quad \langle \varepsilon, \nu \rangle \Downarrow n \quad n \neq 0 \quad \varphi' = \nu \;\texttt{\&\&}\; \varphi}{\langle \sigma, \varepsilon, \texttt{if } e \texttt{ then } c_t \texttt{ else } c_f, a, \varphi :: s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma, \varepsilon, c_t, a, \varphi' :: s, n, \mathcal{E} \rangle} \text{ SC}^+\text{-BR-T}$$

$$\frac{\langle \mathcal{E}, e \rangle \downarrow \nu \quad \langle \varepsilon, \nu \rangle \Downarrow 0 \quad \varphi' = \; !\nu \;\texttt{\&\&}\; \varphi}{\langle \sigma, \varepsilon, \texttt{if } e \texttt{ then } c_t \texttt{ else } c_f, a, \varphi :: s, n, \mathcal{E} \rangle \Rrightarrow \langle \sigma, \varepsilon, c_f, a, \varphi' :: s, n, \mathcal{E} \rangle} \text{ SC}^+\text{-BR-F}$$